NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

THE USE AND RUN-TIME OVERHEAD OF CORBA IN MSHN PROJECT

by

Alpay Duman

September 1998

Thesis Advisors: Debra Hensgen
Ted Lewis

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1998	Master's Thesis

4. TITLE AND SUBTITLE: THE USE AND RUN-TIME OVERHEAD OF CORBA IN MSHN PROJECT

5. FUNDING NUMBERS

6. AUTHOR(S) Duman, Alpay

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT:
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE:

13. ABSTRACT *(maximum 200 words)*

The goal of the Management System for Heterogeneous Networks (MSHN) is to support the execution of multiple, disparate, adaptive applications in a dynamic, distributed heterogeneous environment. MSHN consists of multiple, eventually replicated, distinct distributed components that themselves execute in a heterogeneous environment. This thesis answers the question: Is the performance of the Common Object Request Broker Architecture (CORBA) sufficient to support MSHN's inter-component communication?

This research focuses on the applicability of communication mechanisms from the CORBA 2.2 specification to MSHN. After a careful literature search, we identified four mechanisms for further examination: the Static Invocation Interface (SII), the Dynamic Invocation Interface (DII), the Typed Event Service and the Untyped Event Service. Our rationale for selecting these mechanisms includes scalability, flexibility, extensibility, portability, maintainability, and manageability for the MSHN system.

We implemented a prototype of MSHN's communication infrastructure using these four mechanisms, and measured their run-time performance. The overhead added by CORBA for distributed component communication of MSHN system varied from a low of 10,6 milliseconds per service request to a high of 279.1 milliseconds per service request on UltraSparc10 boxes with Solaris 2.6 Operating System and connected via 100 Mbits/sec Ethernet. We therefore conclude that using CORBA mechanisms will not only substantially decrease the amount of time required to implement MSHN, but if used appropriately they will not substantially degrade performance.

14. SUBJECT TERMS CORBA, distributed computing, performance overhead			15. NUMBER OF PAGES 93
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Approved for public release; distribution is unlimited

THE USE AND RUN-TIME OVERHEAD OF CORBA IN MSHN PROJECT

Alpay Duman
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1998

Author: Alpay Duman
Alpay Duman

Approved by:
Debra Hensgen, Thesis Advisor

Ted Lewis, Thesis Advisor

Dan Boger, Chairman
Department of Computer Science

ABSTRACT

The goal of the Management System for Heterogeneous Networks (MSHN) is to support the execution of multiple, disparate, adaptive applications in a dynamic, distributed heterogeneous environment. MSHN consists of multiple, eventually replicated, distinct distributed components that themselves execute in a heterogeneous environment. This thesis answers the question: Is the performance of current implementations of the Common Object Request Broker Architecture (CORBA) sufficient to support MSHN's inter-component communication?

This research focuses on the applicability of communication mechanisms from the CORBA 2.2 specification to MSHN. After a careful literature search, we identified four mechanisms for further examination: the Static Invocation Interface (SII), the Dynamic Invocation Interface (DII), the Typed Event Service and the Untyped Event Service. Our rationale for selecting these mechanisms includes scalability, flexibility, extensibility, portability, maintainability, and manageability for the MSHN system.

We implemented a prototype of MSHN's communication infrastructure using each of these four mechanisms and measured their run-time performance. The overhead added by CORBA for distributed component communication of the MSHN system varied from a low of 10.6 milliseconds per service request to a high of 279.1 milliseconds per service request on UltraSparc10 boxes running the Solaris 2.6 Operating System and connected via 100 Mbits/sec Ethernet. We therefore conclude that using CORBA mechanisms will not only substantially decrease the amount of time required to implement MSHN, but if used appropriately they will not substantially degrade performance.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank to my mother Mubeccel Duman for her support throughout my life..

I would like to thank Dr. Debra Hensgen for her help and guidance during my work. I appreciate her endless energy, her desire to explore.

I would also like to thank Dr. Ted Lewis for sharing his life-time knowledge and experience with me.

I also want to express my appreciation to Matthew Schnaid for his contributions to my work.

I would like to thank Sushanna St. John for her help to include the figures.

I. INTRODUCTION

In the Heterogeneous Processing Laboratory at the Naval Postgraduate School, we are designing, implementing, and testing a resource management system called the Management System for Heterogeneous Networks (MSHN). MSHN is part of the Defense Advanced Research Projects Agency (DARPA) sponsored Quorum program. MSHN's goal is to support the execution of multiple, disparate, adaptive applications in a dynamic, distributed heterogeneous environment. To accomplish this goal, MSHN consists of multiple, eventually replicated, distinct distributed components that themselves execute in a heterogeneous environment. These components will have widely varying functionality, will come in and out of existence, will communicate via heterogeneous networks, and will execute on different platforms. These system components are also likely to be written in different programming languages. We can, of course, at the expense of a great deal of programmer's time, implement from scratch, specialized naming services to locate the appropriate component at run-time and specialized communication mechanisms to enable communication between these heterogeneous platforms. Alternatively, we can use a general tool, such as Common Object Request Broker Architecture (CORBA), to achieve the same functionality while reducing the development time. Experience with generalized systems such as CORBA, has revealed that the reduced development time cost comes at the expense of run-time performance, which can be critical, in real-time applications. This thesis, therefore, investigates the utility and overhead of communication mechanisms from the CORBA 2.2 specification to support MSHN's inter-component communication.

To build MSHN's communication infrastructure, we identified four mechanisms from CORBA 2.2 specification for run-time performance examination: the Static Invocation Interface (SII), the Dynamic Invocation Interface (DII), Typed Event Service, and Untyped Event Service. After settling on these four mechanisms, we implemented a prototype of MSHN's communication infrastructure using each of them,

and measured their respective run-time overhead.

A. MOTIVATION

In this section, we discuss the current computing environment of the Department of Defense (DoD). Observing this environment, we can understand the economical and technical reasons for DoD's move towards COTS. After defining the concepts, **open systems**, **industry standards**, and **middleware**, we review the relationships between them. Additionally, we describe the effects of these concepts on the information technology market. We conclude the section by explaining why we examined middleware, i.e., CORBA, for our MSHN architecture.

DoD has a great amount and variety of computing hardware. Some of this hardware was designed to run a particular set of application programs, while other hardware is general purpose in nature and may run a broad variety of applications. Many of these physically dispersed machines are connected via networks and the Internet, and therefore make up a distributed, heterogeneous computing base [Ref. 1]. With decreasing defense budgets and a vast, connected, heterogeneous hardware base, the challenge is to maximize the throughput of the jobs using the existing heterogeneous, distributed resources while minimizing the cost for interconnecting these resources without compromising the DoD specific requirements (e.g., run-time performance, reliability, and accuracy of the systems). Additionally, systems and applications should be developed so that they can easily be extended to fulfill future needs. In particular, a major DoD goal is to enable the warfighter to exchange classified and unclassified, tactical and non-tactical, information across platforms at shore and at sea. These exchanges should appear to be seamless to the actual user and should not create any new proprietary systems or require additional costs for research and development (R&D). Therefore, DoD, and especially the Navy, looks toward commercial-off-the-shelf (COTS) software to address this challenge.

There are a number of reasons to use COTS in the DoD environment, even

though such use was uncommon just a few years ago. The primary reasons for moving towards COTS are the low initial purchase cost and the short development time. With downsizing, and decreasing defense budgets, DoD can no longer substantially subsidize the computer industry. COTS reduces the amount of time and money required for software development projects. Another reason for moving towards COTS is that they provide more interoperability if they follow industry standards. In an environment where COTS products facilitate seamless interoperability, large system designs will evolve more easily and quickly.

A major direction of COTS today is towards enterprise computing, open systems, and industry standards. Here, the term enterprise refers to a reasonably large organizational unit, i.e., a service branch of the Armed Forces, the entire DoD, or a large branch of a company. From an information technology (IT) point of view, a typical enterprise consists of a wide variety of different, almost always multi-vendor, hardware and software running different applications for different domains. Examples of military domains include acquisition, word processing, intelligence, and weather forecasting.

Today, corporate strategists favor linking all of their resources into a single enterprise that will enable all of their employees to share not only the resources, but also information. Additionally, they want to extend their businesses by linking their enterprise-wide systems with those of their business partners, suppliers, distributors, and customers. To interconnect all of these systems, and to move data from one system to another, these multi-vendor systems must seamlessly interoperate with one another. Furthermore, any new application must be designed and implemented in such a way that its development will not allow a particular hardware or software vendor to obtain a monopoly and hence, prevent the users from taking advantage of truly new technology.

Open system applications are, by definition, intended to be source code portable across platforms in order to prevent such problems. In addition, as the applications

and the systems grow in scope, the IT department of an enterprise must be able to move the applications or the information to a different solution platform without much difficulty. On the other hand, the benefits of open systems cannot be attained without industry standards. Organizations that are dependent upon one vendor's solutions and who are trying to alter

- any of the vendor's methods,
- the management or operating system, or
- the hardware architecture,

will do so only at great expense. Generally, enterprises that have utilized proprietary solutions have invested a large amount in applications, documentation, and trained employees, for an existing proprietary system. Any change in these systems will cause loss of most of these investments, and will also require time to implement new systems to replace the old ones. Today, the challenges for CIO's, who are currently using proprietary systems, are (1) when and if they should move to a new system, and (2) whether they should choose a less expensive vendor specific solution, or generally more expensive open system solution.

" The battlefield is a scene of constant chaos. The winner will be the one that best controls chaos, both his own and that of his enemy."[Napoleon Bonaparte] The IT market is in chaos. The vendors are absorbing generic industry standards and extending them with new non-standard features to force the customers to depend on their product for particular domain solutions. In this case, the customers work hard to prevent dependence upon vendor specific solutions." Therefore, determine the enemy's plans and you will know which strategy will be successful and which will not."[Sun Tzu] Hence, the military must recognize the vendors' strategies and respond to them accordingly. The military might benefit from considering this situation as a two-player, zero-sum, non-cooperative game. The vendors are trying to increase their market share, whereas the customers (including the military) must ensure that

their computing solutions are flexible. Game theory tells us that the best the player can do is to choose either the dominant strategy equilibrium or the Nash equilibrium. The first is a player choosing the best action that is a response against any action the other might take (fire-look). The latter is a player choosing the best action that is a response to the action the other takes (look-fire). In this game, the vendors and the customers encounter one another in a repeated environment. This environment introduces two important new elements into the game. First, players can think in terms of contingent strategies. The customers' decision may depend on the history of the computer industry. Second, in repeated play the present is not the only thing which affects the decision. Especially, in the computing industry the future is as important as the present. [Ref. 2]

An alternative to open solutions is third party software. Although third party software might be inexpensive, organizations have found that with this solution, they not only are still forced into using proprietary systems, but also they lose much of the competitive edge provided by IT.

Before we narrow our focus in open systems to middleware, which is the glue supporting interoperability within and between distributed systems, we would like to differentiate a software architecture from a software product. A software architecture consists of a set of definitions, rules, and terms that are used as guidelines to build a product. A product, on the other hand, is a specific implementation of an architecture by a vendor. Open architectures are often based on industry standards, so that they can survive the economical and technical lifetime of more than one product and can themselves form the basis of a new standard to satisfy evolving needs. [Ref. 3]

Since one of our objectives in this thesis is the interoperability and portability of MSHN in heterogeneous, distributed networks, we will focus on middleware. Conceptually, middleware is the glue between the system components in a heterogeneous environment. When a client component uses a pre-defined application-programming interface (API) to invoke a service over a network, the middleware transmits the

client's request over the network to the server component. It is also responsible for conveying the resulting response back to the client component. In this paradigm, the components are not aware of their different platforms and different data representations. Thus, middleware helps to integrate system components across a distributed, heterogeneous environment. In particular, middleware, while distributing the processing load among multiple heterogeneous systems, should allow components executing on very different platforms to interact and to share resources and information.

The goal of military information technology for the 21st century is shifting from platform centric to network centric warfare. We need the ability to get real-time information to the warrior in the field who is using a palm-top, and to the warrior at sea who is using a desktop PC. In many instances, the information needed by the warrior must be obtained from an application running on a computer with more power than the one that the warrior has in hand. In this case, results must be sent to the warrior's machine. This heterogeneity of the mission-critical military applications and the need for the interoperability motivates the use of middleware.

B. SCOPE OF THESIS

This research focuses on the applicability of communication mechanisms from the CORBA 2.2 specification to MSHN. MSHN is a resource management system for heterogeneous networks. MSHN addresses the challenge of supplying quality of service for the adaptive, mission-critical applications in a distributed, heterogeneous environment. MSHN itself consists of distributed components residing in a heterogeneous environment to accomplish its goal. Thus, we looked into the communication mechanisms of CORBA. After a careful literature search, we identified four mechanisms for further examination: the Static Invocation Interface (SII), the Dynamic Invocation Interface (DII), Typed Event Service and Untyped Event Service. Our rationale for selecting these mechanisms includes scalability, flexibility, extensibility, portability, maintainability, and manageability for the MSHN system. After settling

on these four mechanisms, we implement a prototype of MSHN's communication infrastructure using each of them, and measure their respective run-time overhead.

C. ORGANIZATION

This thesis is organized as follows: Chapter II overviews MSHN's client/server architecture. Chapter III discusses relevant sections of CORBA 2.2 specification. It introduces OMG's Object Management Architecture (OMA) Reference Model, describes CORBA's method invocation mecanisms and reviews two CORBA services used in this research: the Naming and Event Services. Chapter IV describes the designs of our four prototypes. In particular, it covers the problems we encountered with CORBA communication mechanisms and our solutions. It also discusses the methodology of the tests we used to determine the overhead added using CORBA and reports the results of those tests. In the final chapter, we summarize our results, the lessons we learned, and we propose future work.

II. MSHN'S CLIENT/SERVER ARCHITECTURE

In this chapter, we will briefly introduce MSHN (the Management System for Heterogeneous Networks) and discuss MSHN's client/server architecture. We will also enumerate the components of MSHN and define briefly the interaction between these components.

A. PURPOSE

In the Heterogeneous Processing Laboratory at the Naval Postgraduate School, we are designing, implementing, and testing a resource management system called the Management System for Heterogeneous Networks (MSHN[1]) which is supported by the Defense Advanced Research Projects Agency's (DARPA) QUORUM program. MSHN's goal is to support the execution of multiple, disparate, adaptive applications in a dynamic, distributed heterogeneous environment. To accomplish this goal, MSHN consists of multiple, eventually replicated, distinct distributed components that themselves execute in a heterogeneous environment. These components will have widely varying functionality, will come in and out of existence, will communicate via heterogeneous networks, and will execute on different platforms.

B. MSHN'S PROPOSED ARCHITECTURE

The MSHN architecture, and the Common Object Request Broker Architecture (CORBA) are both still evolving. This thesis studies the feasibility of using mechanisms from CORBA's latest version, CORBA 2.2, to facilitate the interactions between the MSHN components. Although we expect that the final design of MSHN may differ slightly from the current one, we do not anticipate the need for additional

[1]Pronounced "mission"

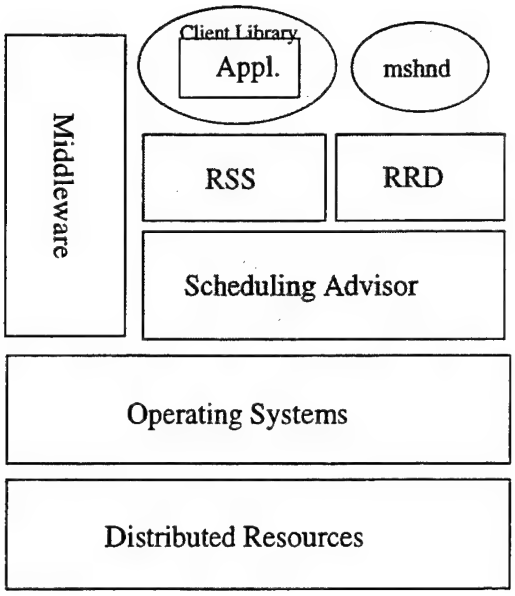


Figure 1. MSHN Conceptual Architecture

components. MSHN architecture consists of multiple instantiations of each of the components enumerated below:

- a Client Library (one for each executing application to be managed by MSHN),
- a Scheduling Advisor (hierarchically replicated),
- a Resource Requirement Database (hierarchically replicated),
- a Resource Status Server (hierarchically replicated), and
- a MSHN Daemon (one for each computing resource).

This thesis analyzes the communication overhead between the system components that is due to the use of generalized middleware, CORBA. Figure 1, the MSHN Conceptual Architecture, shows all of the components, which are shaded, as **translucent layers** executing on distributed platforms. A translucent layer is one that can be bypassed by layers that are above or below it. For example, the MSHN Daemon (mshnd) can interact directly with the operating systems, bypassing all of the Resource Status Server, the Resource Requirement Database and the Scheduling Advisor. In the environment that MSHN supports, both MSHN and non-MSHN

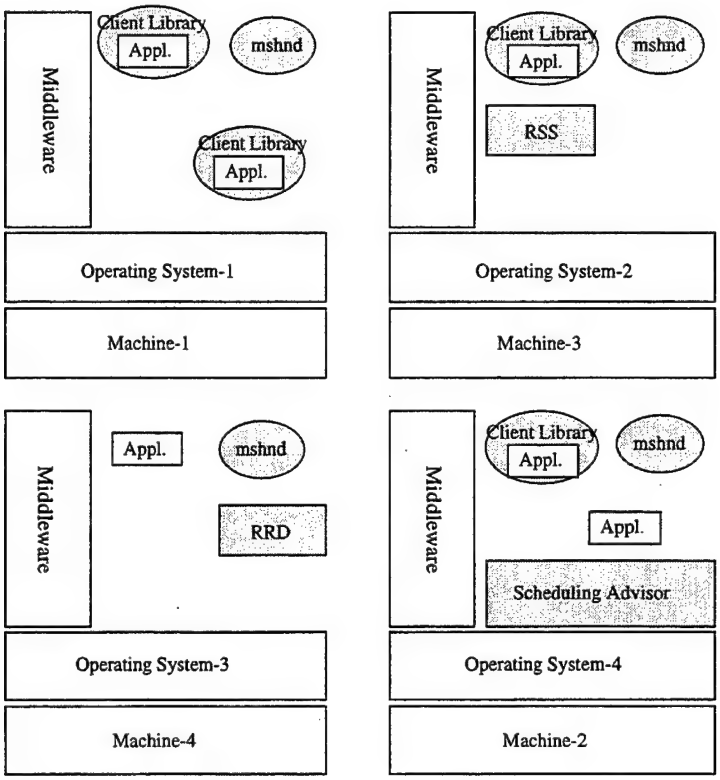


Figure 2. Example MSHN Physical Instantiation

applications may be executing at any given time. Figure 2 illustrates how these components might actually be distributed among different heterogeneous machines. CORBA mechanisms, particularly the Internet Inter-ORB Protocol (IIOP), can be used to facilitate communication between components.

Because this thesis investigates facilitating communication between the components, the MSHN description in the remainder of this chapter emphasizes the interactions between the components. We have included brief descriptions of the functionality of each of the components in Appendix B. For further information, the reader may refer to a technical report describing the entire project [Ref. 4].

Figure 3, MSHN's Software Architecture, illustrates all of the interactions between the system components. MSHN has a peer-to-peer architecture. In peer-to-peer architecture, the client can request a service from any server in the system, i.e. the MSHN Daemon must not go through the Scheduling Advisor to update the

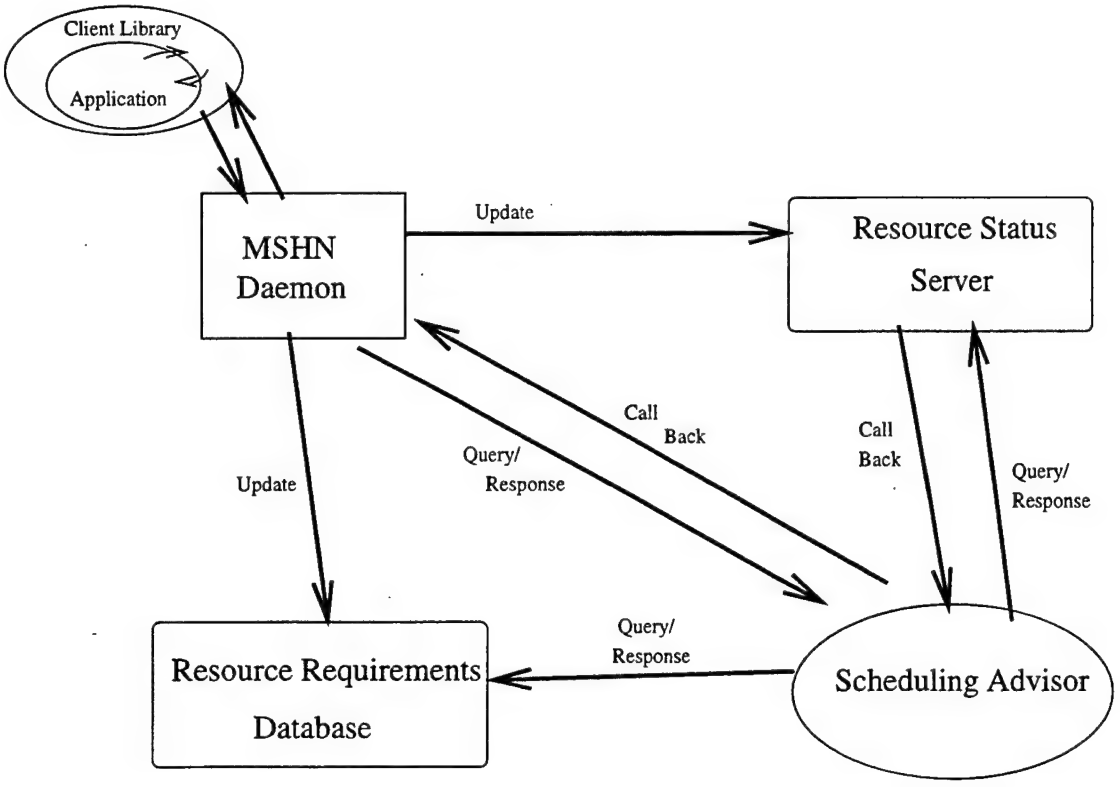


Figure 3. MSHN's Software Architecture

Resource Status Server or Resource Requirement Database. [2]

We now present two-tier and three-tier views to give a clear understanding of the interactions between the system components. Typically, many applications, each linked with the MSHN Client Library will be running at any given time. They will need to communicate with a Scheduling Advisor (SA), possibly via the MSHN Daemon, to request appropriate resources for starting new processes. They may also communicate with a MSHN Daemon when receiving their recommended schedule. Additionally, the Client Libraries update the Resource Requirement Database (RRD) and the Resource Status Server (RSS) with the expected resource requirements of the applications, and current resource availability within the MSHN system. Figure

[2]In distributed systems, callbacks are useful in supplying asynchronous communication. Callbacks transmit notification of events without blocking the event originator. Callbacks flow from the servers towards the clients. When callbacks are used the client and the server have a peer-to-peer relationship.

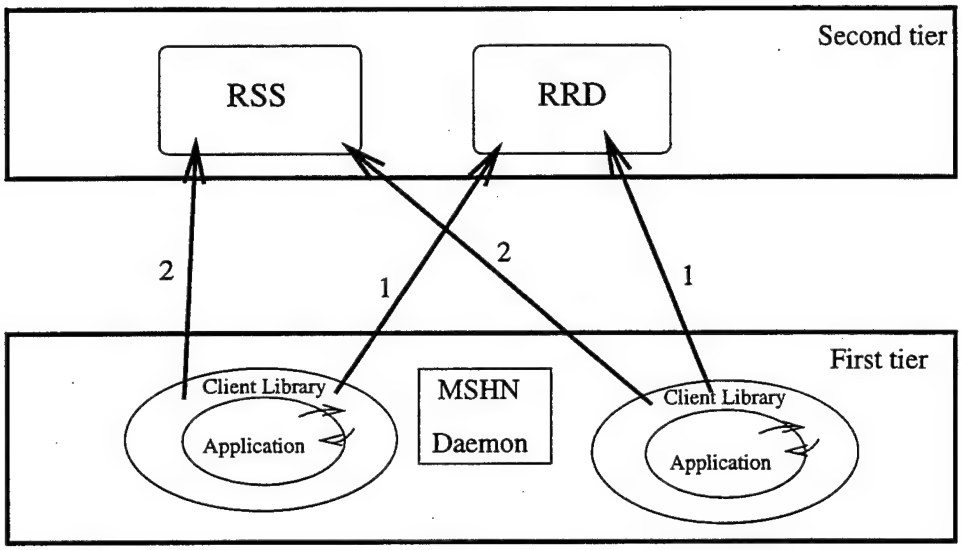


Figure 4. Two-tiered Architectural View of MSHN Architecture

4 illustrates this updating interaction as a two-tiered client/server architecture. The update frequency of the Resource Status Server is expected to be high so that it, in turn, can supply the Scheduling Advisor with accurate, current information.

We anticipate that the frequency of the updates will cause excessive network load and a considerable processing load on the Resource Status Server and the Resource Requirement Database. To avoid these loads, MSHN's design includes proxy Resource Status Servers and Resource Requirement Databases that will come in and out of existence, as they are required to minimize the number of updates. These proxies will filter gathered information and update the hierarchical Resource Status Server and the hierarchical Resource Requirement Database when necessary.

The Scheduling Advisor resides between the information needed to schedule (the Resource Status Server and the Resource Requirement Database) and the requesters of schedules (applications linked with the Client Library), which indicates that there will be a high communication rate to and from the Scheduling Advisor. We can therefore also view MSHN as having three tiers, where the Scheduling Advisor is the second tier, and the Resource Status Server and the Resource Requirement Database are in the third tier (see Figure 5). When the MSHN Daemon contacts the

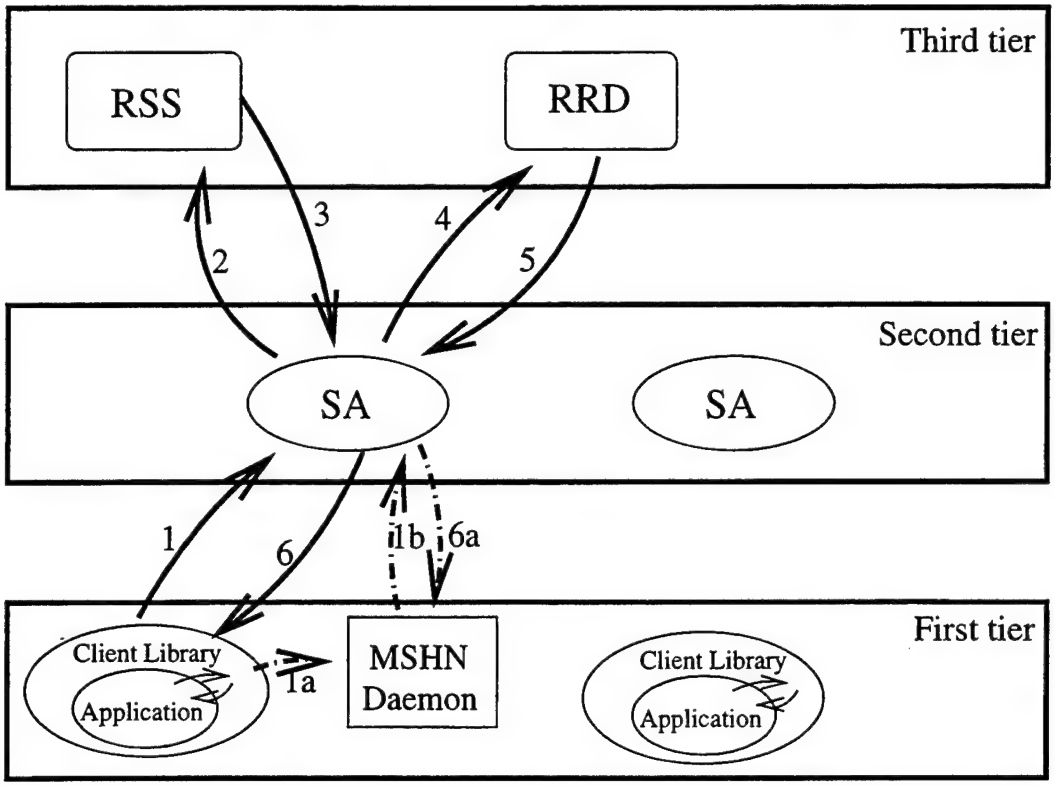


Figure 5. Three-tiered View of MSHN

Scheduling Advisor on behalf of a Client Library for a schedule, the Scheduling Advisor queries both the Resource Status Server, and the Resource Requirement Database before it computes the schedule and sends it to the MSHN Daemon.

Although the Client Libraries are the initiators of many of the communication chains through the MSHN system, other chains are initiated by the Resource Status Server. In the case of a violation of a deadline because of a change in resource availability, for example, the Resource Status Server will trigger the Scheduling Advisor to reschedule a process that would not otherwise meet its deadline. The Scheduling Advisor will adapt to the new situation by either changing the format of the process or restarting it on a different resource via the MSHN Daemon. This interaction is the reverse of the previously described communication chain. Figure 6 shows this three-tiered view.

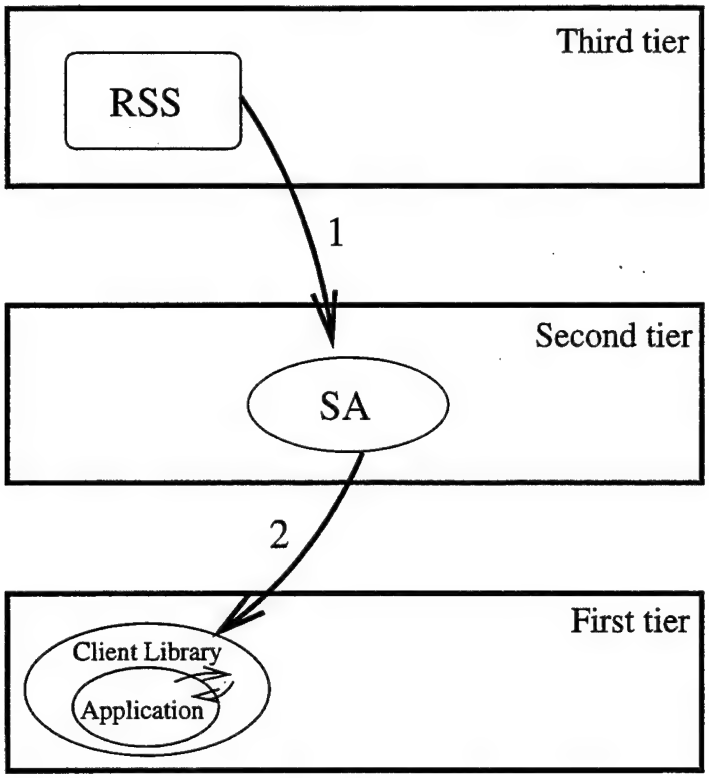


Figure 6. Alternate Three-tiered View of MSHN

Although we have shown several two and three tier views of MSHN, the reader should understand that these are only examples. Much larger chains will actually exist when the various components are hierarchically replicated.

C. SUMMARY

MSHN and CORBA are both still evolving. MSHN's goal is to deliver end-to-end quality of service to adaptive, mission-critical applications in a dynamic, distributed, heterogeneous environment. To accomplish this goal, MSHN consists of multiple, eventually replicated, and distributed components that must frequently communicate through heterogeneous networks.

In the next chapter, we supply some background concerning the CORBA 2.2 specification so that the reader can understand why CORBA is a viable candidate for supporting these interactions.

III. OVERVIEW OF CORBA

CORBA is an evolving, open standard, which is regulated by Object Management Group (OMG) to bring some order to the rapid and disjoint development of object technologies. The OMG is a coalition of over 900 companies, some of which are system developers, and others are users. The OMG's main objective is to influence the object technologies. They define the Object Management Architecture (OMA) Reference Model, upon which all OMG specifications are based. CORBA, the most commonly used OMG specification, supports the construction and integration of object-oriented software components in heterogeneous distributed environments. In this chapter, we provide a brief overview of the CORBA 2.2 specification to give a basic understanding to the reader. In the next chapter, we provide a rationale for our design decisions concerning the use of CORBA for implementing the prototype of the MSHN client/server architecture. This overview includes definitions of some elements from the OMA Reference Model to familiarize the reader with the terminology. Additionally, it describes two method invocation mechanisms from the specification, as well as two CORBA services, the Naming Service and the Event Service, that we used in our prototypes. For further information, the reader may refer to OMG documentation and other references listed in the bibliography of this thesis [Ref. 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Any reader familiar with the CORBA 2.2 specification may skip this chapter.

A. THE OBJECT MODEL

Understanding the meaning of the term **object**, as it is used in CORBA specifications, is fundamental to the understanding of CORBA. Although the term object is widely used in computer science, this section reviews its exact semantics as defined in OMG's OMA Reference Model.

1. Objects in OMA

An application consists of one or more objects. These objects may reside on the same or different platforms. An object enforces encapsulation, polymorphism, inheritance, and persistency. It provides one or more services that can be requested by a **client**. The same object can serve more than one client, concurrently. An **object reference** is a value that denotes a unique object. Specifically, an object reference will identify the same object each time the reference is used in a request (subject to certain pragmatic limits of space and time).

If a CORBA user wants an object to be accessible through multiple CORBA vendors, the ORB, on behalf of the user, must assign a unique **interoperable object reference** to that object. For example, a Java client, running on top of IONA Technologies OrbixWeb, must be able to invoke a C++ object, running on top of Inprise (formerly Borland) VisiBroker for C++, independent of the manner in which these different vendors assign object references to objects that are accessed only within either domain.

2. Requests

Clients obtain services from an object by making **requests**. A request consists of (1) an operation, (2) the name of the object that will respond to the request, (3) zero or more (actual) parameters, and (4) an optional request context. The object that will perform the operation on behalf of the client is called the **target object**. One possible outcome of performing a service on behalf of the client is returning to the client the results, if any, defined for the request. If an abnormal condition occurs during the servicing of a request, an exception is returned.

3. Interface

Associated with each object is (at least) one **interface**. A client can request any operation specified in an interface. An interface also specifies exceptions that may occur during these operations and type definitions required by the client to

invoke these operations on the object. An object implements an interface if it can be specified as the target object for each operation described by the interface.

4. Operation

An **operation** denotes a service that can be requested from an object. As traditional functions in any programming language, an operation has a signature that describes the parameters of, and results returned from, that operation. In particular, a signature consists of:

- a specification of the parameters required in requests for that operation;
- a specification of the type of the result from the operation;
- an identification of the user exceptions that may be raised by a request for the operation;
- a specification of additional contextual information that may affect the request; and
- an identification of the execution semantics that the client should expect from a request for the operation.

Code that is executed to perform a service is called a **method**. A method defines the implementation details of an operation.

5. Execution Semantics

Two styles of execution semantics are defined by the object model.

- **At-most-once**: In this model, an object can either return successfully or return an exception. If an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed no more than once.
- **Best-effort**: A best-effort operation may not have any output parameters, any input/output parameters, and may not return any result. The client continues immediately after making the request and never synchronizes with the completion of the request. If an exception occurs on the object side, the client will not be notified of it.

B. THE ARCHITECTURE

In this section, we motivate the use of remote invocation and outline two different semantics for it. We also define the invocation mechanisms within CORBA, i.e. Static Invocation and Dynamic Invocation. We show how these invocation mechanisms are mapped into the semantics. Finally, we study the components of CORBA, including their different responsibilities that are used to provide CORBA's invocation mechanisms.

For simplicity in system development, clients will get some services that they need from other objects. When an object and its client reside on the same host, they will likely exist in different address spaces. Therefore, a client can no longer use local invocation to obtain services. Additionally, the user's application may demand more computing resources e.g., memory or CPU, than the local host has. In that case, the client needs to go not only across the address space but also across the network. To execute a method invocation across address spaces, or across a network, we require a new mechanism, **remote invocation** (see Figure 7). Since clients and object implementations are no longer in the same address space, CORBA defines an **Object Request Broker (ORB)**, which is a well-known point of contact for both client and object implementations (see Figure 8). The client invokes an operation on any object through the ORB, which supports the client by providing transparency of object location, transparency of object implementation, transparency of object execution state, and transparency of communication mechanism between the client and the object.

To facilitate communication between the clients and the objects, CORBA supports two types of invocation semantics, **synchronous invocation** and **asynchronous invocation**. Synchronous invocation is blocking. Using this type of invocation, the client will invoke the method and block until it receives a response from the object implementation. Asynchronous invocation is non-blocking. The client will invoke the method, continue its computation, and collect the results when they ar-

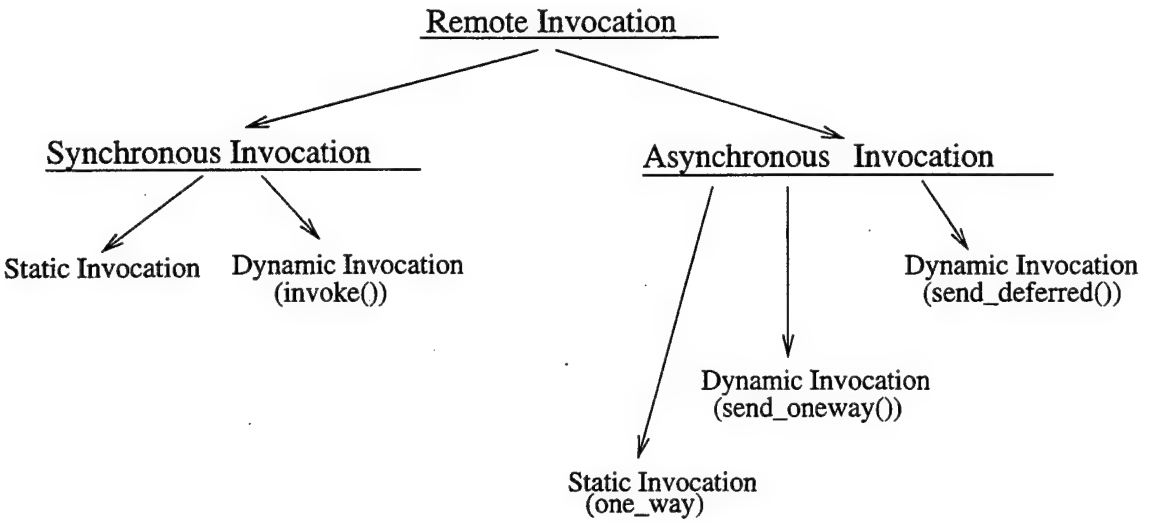


Figure 7. Remote Invocations

rive. Figure 7 also illustrates how CORBA communication mechanisms and their semantics map into these types of invocation.

All objects can obtain a reference to the ORB running on their machine by calling CORBA's `ORB_init()` function. To request a service, clients must furnish an

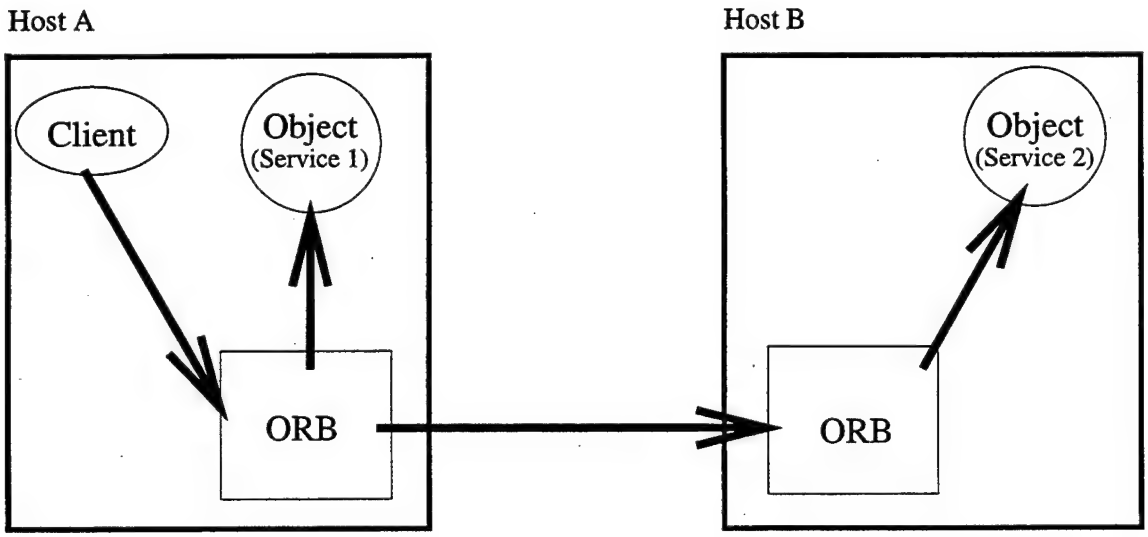


Figure 8. The Clients and Objects

object reference to the Object Request Broker (ORB). The clients may obtain object references in three different ways.

- A client may obtain an object reference from a CORBA Service, such as the Naming Service. The Naming Service simply looks up a name and returns the reference associated with that name. We discuss the Naming Service in more detail at the end of this chapter.
- When clients instantiate new objects remotely, they receive the object reference as a return value.
- Finally, clients may use object references that are stored in persistent storage. Object references in persistent storage are **stringified**. To stringify an object a client invokes the method `object_to_string()` from the ORB. It then stores this either in a file or a database for future use. When the client needs to invoke a method on that object, it can get that string back from the persistent storage and reconvert it to an object reference by invoking `string_to_object()` from the ORB.

The object implementation supplies data for the objects instance and source code for the methods of the object. Often the object implementation will use other objects and private methods to implement its functionality. Object implementations may be written in a variety of languages (e.g., C, C++, Ada, Java, or SmallTalk), and may exist in a variety of forms (e.g., separate servers, libraries, one program per method, an encapsulated application, or an object-oriented database). With additional **object adapters**, it is possible to support virtually any style of object implementation (i.e., a C++ object implementation, an Object Oriented Database object implementation).

Object Adapters are the run-time components of CORBA that sit between the ORB and the Object Implementations. The designers of CORBA could have specified that the ORB should support many different programming styles and languages (see Figure 9). However, this would make the ORB large, complex, and inextensible. An alternative is shown in Figure 10. The alternative allows for new programming styles (and languages) to be supported without changing the ORB and reduces the size and complexity of the ORB.

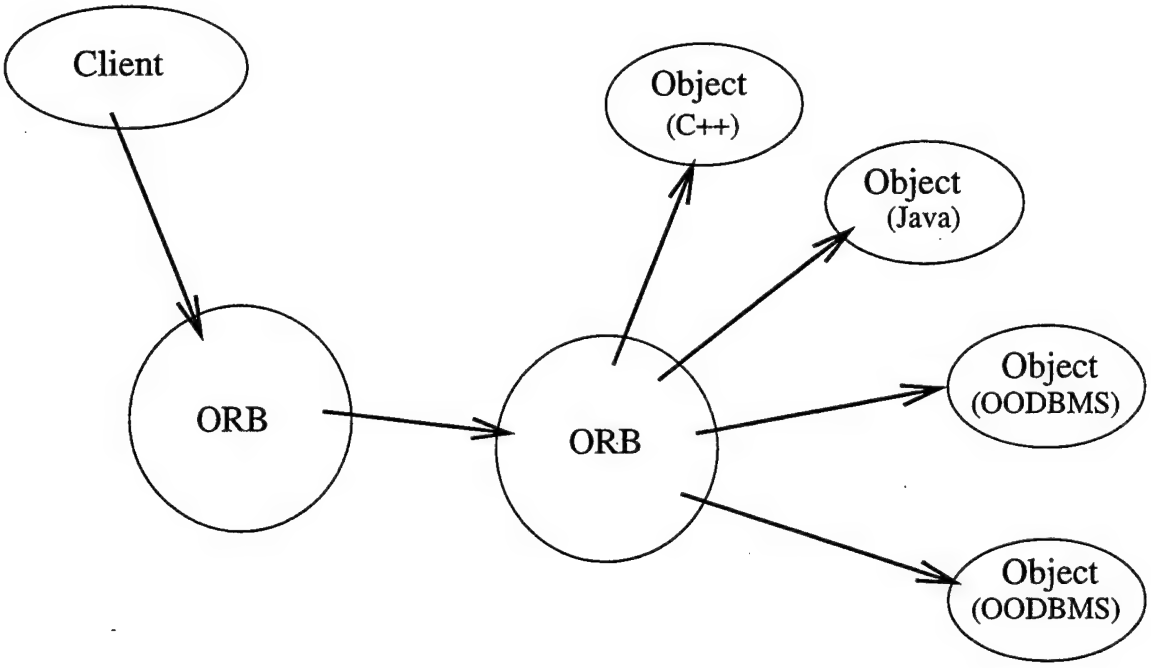


Figure 9. The Relationship between the ORB and the Object Implementation without the Object Adapter.

The **Implementation Repository** is used by the Object Adapter. The Implementation Repository is a run-time repository containing access information about all of the currently available objects. This access information includes the reference for an object, the owner of an object, and a list of users that can launch and invoke operations on that object.

Having understood the motivation for the CORBA components needed by all applications, we now turn our attention to components which are only needed for Static Invocation and those that are only needed for Dynamic Invocation.

1. Static Invocation

If the client is compiled with the interface of an object implementation, the client can invoke that object statically. Static Invocation is easier to use for the application developer because the method invocations in the client's source code are quite similar to local method invocations. Additionally, compile time type checking offers better coverage than run-time type checking. However, Dynamic Invocation

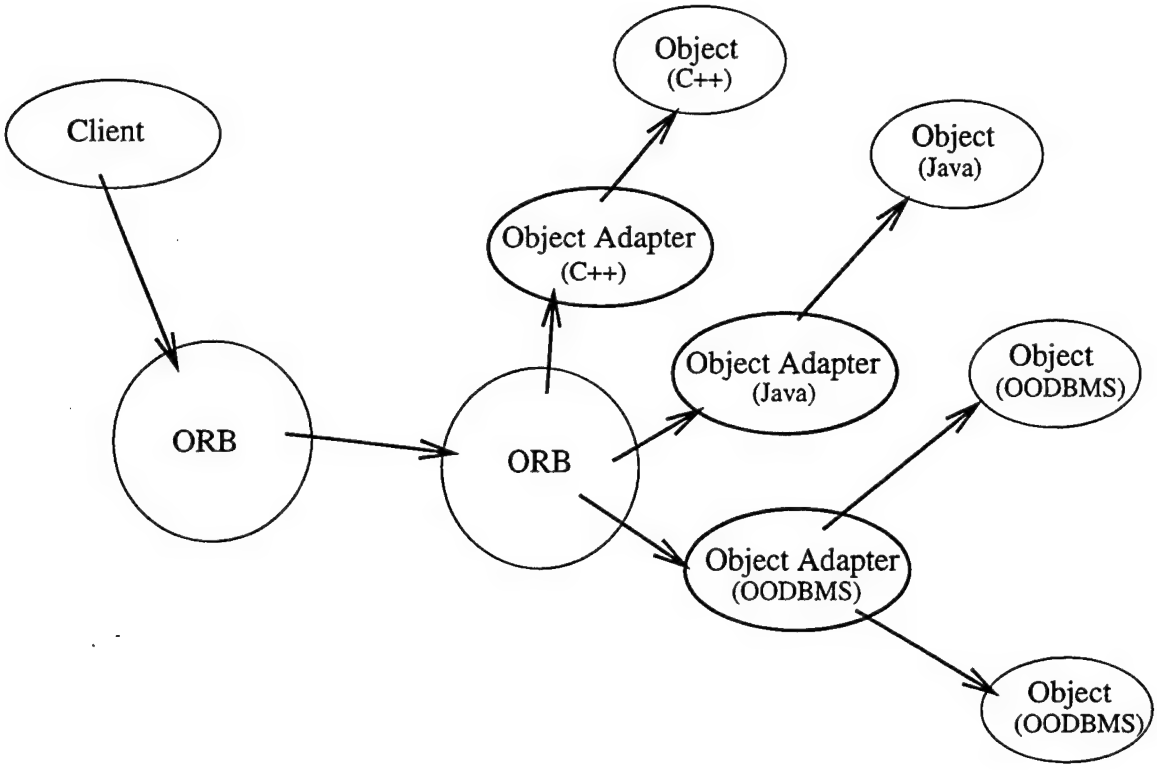


Figure 10. The Relationship between the ORB and the Object Implementation with the Object Adapter.

allows for more flexibility and later binding times, which we will discuss in the next section.

In Static Invocation, the client's platform may still be different from the object's platform. For example, the client may reside on a Windows NT, Intel architecture machine, executing an Orbix ORB, whereas the object may reside on a Solaris 2.6, SunSparc architecture machine, executing a Visibroker ORB. Therefore, Static Invocation must still provide for interoperability across platforms, as well as between vendors. CORBA uses the **Interface Definition Language (IDL)** and **General Inter-ORB Protocol (GIOP)**, in particular Internet Inter-ORB Protocol (IIOP), to provide this functionality. IDL is used to describe the interface of a CORBA object, independent of the object's platform and the programming language. The developer compiles an IDL file with an IDL compiler to generate the

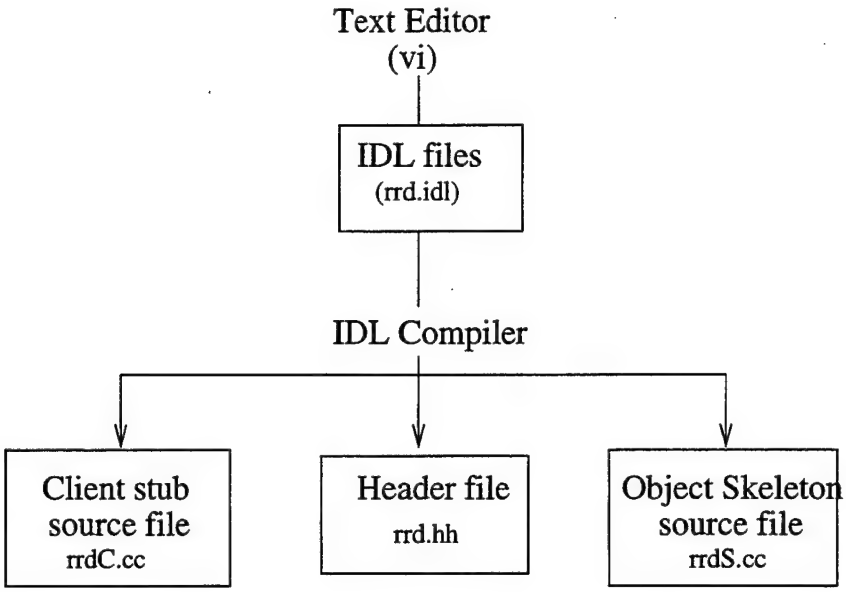
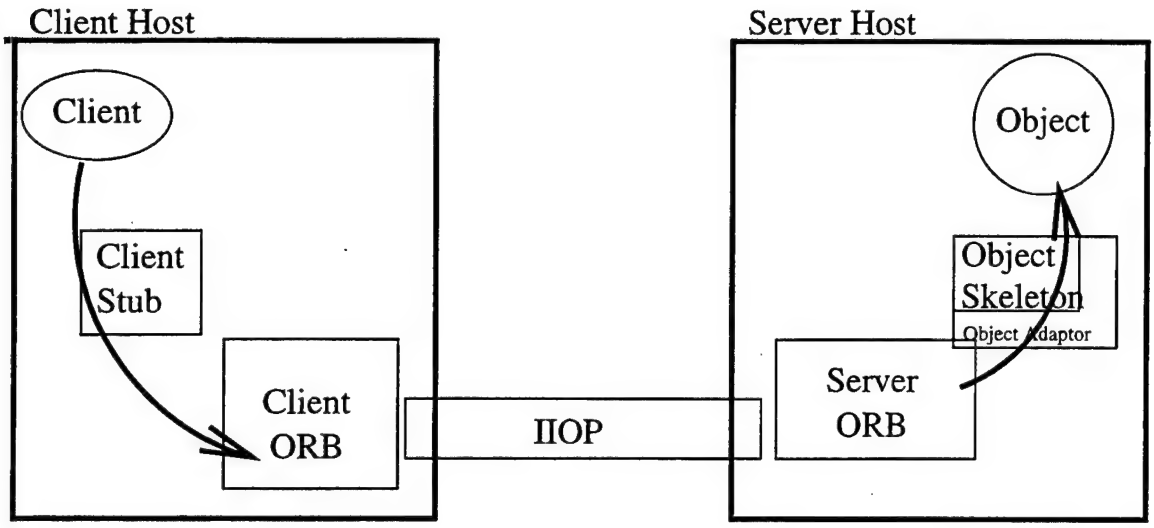


Figure 11. Generating Client Stubs and Object Skeletons

Client Stub, the **Object Skeleton**, and a header file (or header files; see Figure 11) for a particular object interface. The Client Stub converts data from the client's local data representation, which is platform and language specific, to the **Common Data Representation (CDR)**, which is independent of platform and language. On the object's platform, the Object Skeleton executes the reverse operation (See Figure 12). Figure 11 illustrates the process of generating the Client Stub and the Object Skeleton with the IDL compiler.

We define IDL and ORB interoperability in more detail in Appendix C. A sample IDL file and the corresponding output of the IDL compiler, i.e., the Client Stub source code, the Object Skeleton source code, and the header file, can be seen in Appendix D.

A CORBA application requires a client implementation, and an object implementation, with its **servant**. The servant implementation instantiates the object or objects, registers the object with the Object Adapter, and blocks awaiting requests. The developer may designate a period after which, if no requests are received, it will execute. Figure 13 illustrates how these stubs and skeletons are linked with the client



IIOP - Internet Inter-ORB Protocol

Figure 12. Components of Static Invocation

and object source codes at compile-time.

Using Static Invocation, the client can make **synchronous** or **one-way invocations**.

- Synchronous invocations are blocking calls. When a client invokes the request synchronously, it blocks and waits for the response from the server.
- The client, when making one-way invocations, continues to execute while the object processes the request. The client cannot be sure that the server received its call(s). Neither values nor exceptions can be returned to the client.

2. Dynamic Invocation

We now describe Dynamic Invocation. The main difference between Static and Dynamic Invocation is that in Static Invocation the object's interface is linked with the client and in Dynamic Invocation it is not. Therefore, CORBA provides a means for clients to identify desired object interfaces at run-time. The **Interface Repository** supports Dynamic Invocation Interface by maintaining a database of the interfaces of object implementations that can be dynamically invoked. If a client does not have the interface of an object implementation at compile-time, it can obtain the interface at run-time from the Interface Repository. Since all interfaces are derived

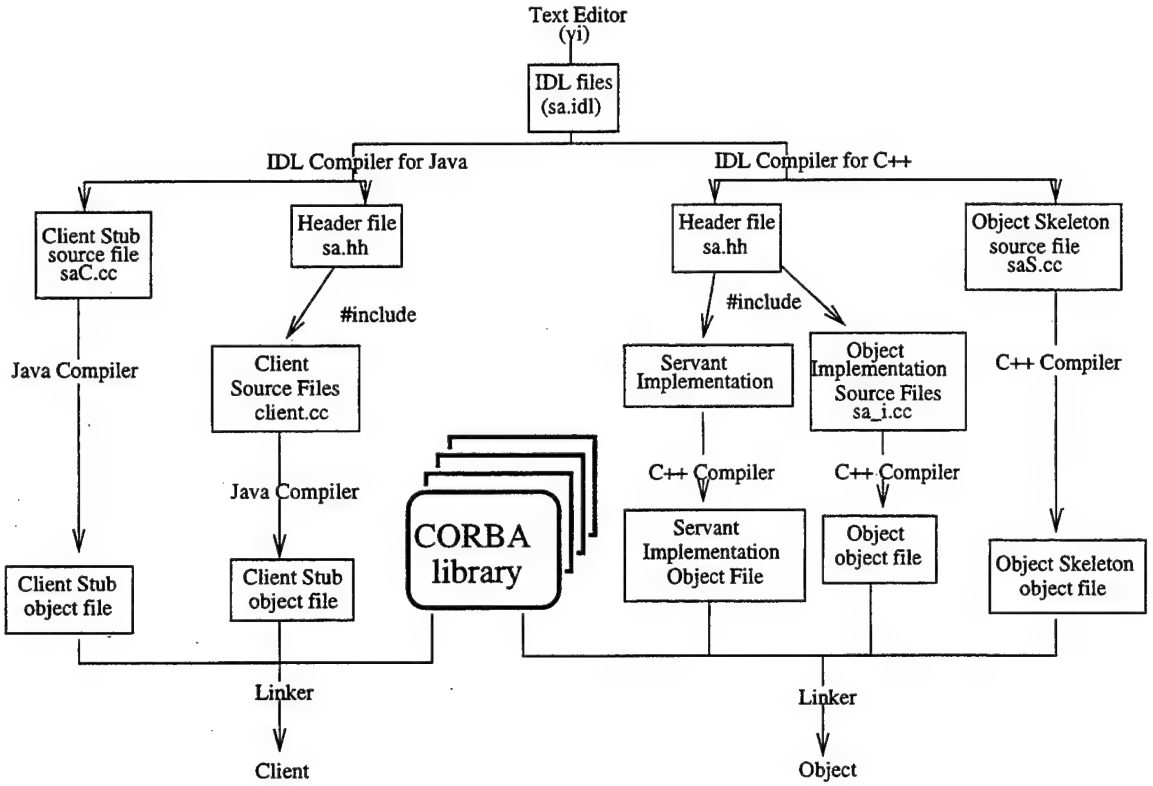
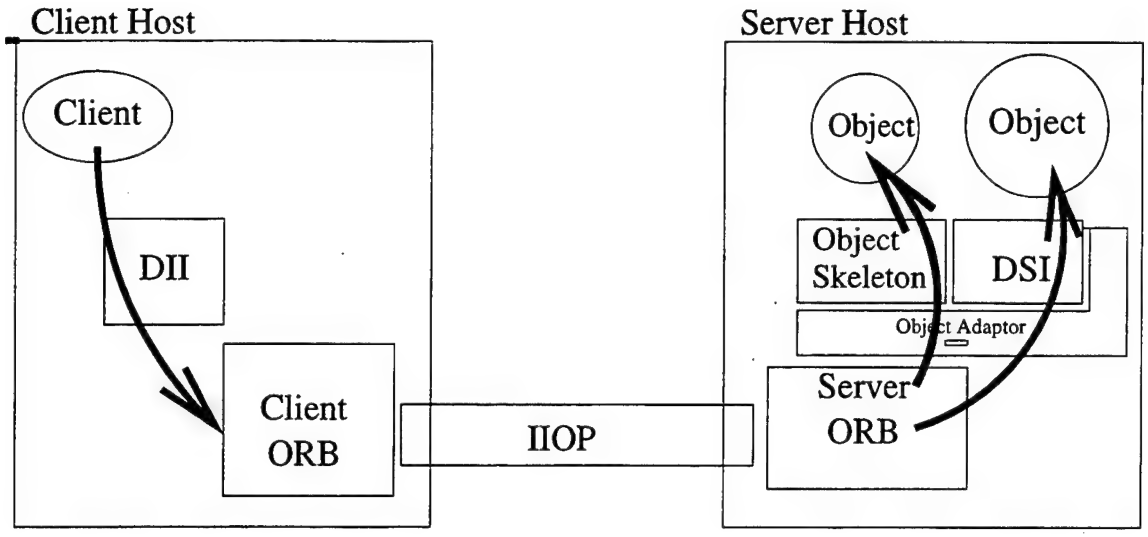


Figure 13. Linking and Compiling

from a generic CORBA object, the client can then use the CORBA object's interface to obtain its own reference. Using this reference, the client can then retrieve information about the interface, i.e., available operations, parameters of those operations, and return types, at run-time.

In the rest of the section, we define only the components required by Dynamic Invocation that are different from the components used for Static Invocation, i.e., the **Dynamic Invocation Interface (DII)** and **the Dynamic Skeleton Interface (DSI)**. The compiling and the linking of the application are the same as they are for Static Invocation.

DII is a generic stub, which is independent of the actual interface of the target object. DII takes the place of the Static Invocation's Client Stub when using Dynamic Invocation (see Figure 14). Additionally, the client may call DII to find the name of an interface, the arguments of a method, and to create a **request object**. A request



DSI - Dynamic Skeleton Interface
DII - Dynamic Invocation Interface
IIOP - Internet Inter-ORB Protocol

Figure 14. Components in Dynamic Invocation

object is a proxy object in the client's address space that appears to the client as the target object. This proxy object handles encoding of the request, and transmitting of the request to the actual target object.

The client uses the DII to create the request object and to populate it with the parameters in the order defined in the operation of the interface. Then, the client invokes a method on the target object by using one of the three invocation mechanisms: Synchronous Invocation, One-way Invocation, or Asynchronous Invocation. We discussed the first two invocation mechanisms in the previous section. The third mechanism, asynchronous invocation, is only available in Dynamic Invocation.

Asynchronous invocations are made without blocking the client, as in one-way invocation. Asynchronous invocation differs from one-way invocation in that both results and exceptions can be returned. After results are returned through the request object, it is released.

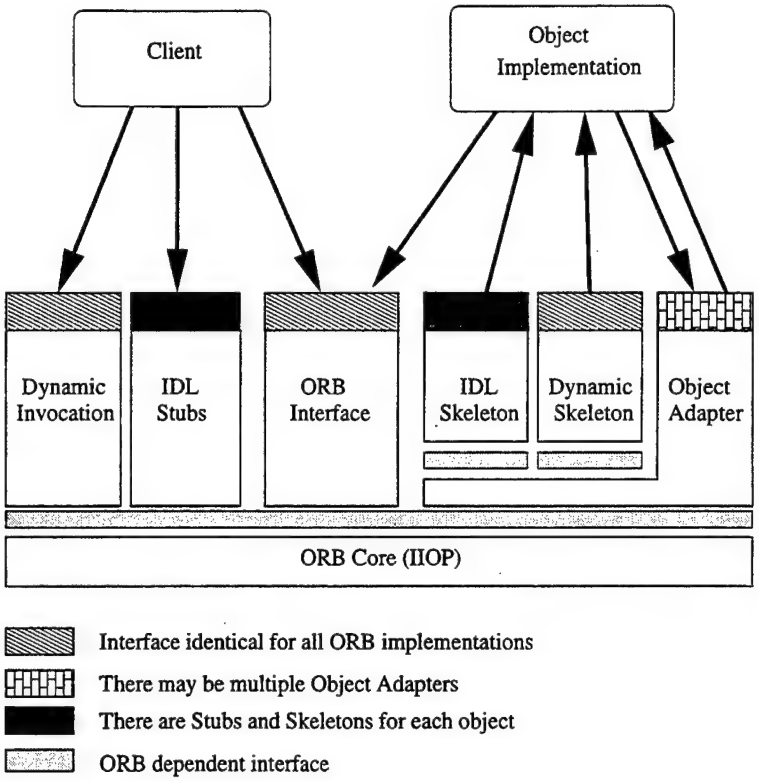


Figure 15. The Architecture

The real advantage of Dynamic Invocation lies in the DII's ability to construct the method invocation dynamically. This type of invocation is useful if the client invokes server objects infrequently or needs to discover them at run-time. However, to discover the information about the interface of the object implementation, the client needs to make several invocations and each of them may be an expensive remote invocation. Therefore, users must be aware of the trade off between the flexibility and the cost of getting all the information about the interface at run-time. We address this issue again in chapter IV and describe a solution that works well in MSHN.

On the server side, the requests go through either the Object Skeleton or the Dynamic Skeleton Interface (DSI). DSI may take the place of the Static Invocation's Object Skeleton. Another advantage of DSI is that it allows for the use of non-CORBA objects. We will conclude the discussion about CORBA components by illustrating the overall architecture and the relations we have defined so far (see

Figure 15).

C. SERVICES

CORBA Services augment and complement the functionality of the ORB. They are collections of system-level services packaged with IDL-specified interfaces. The use of these services is optional, and they are extremely generic. In this section, we discuss two of the most widely used CORBA Services.

1. CORBA Event Service

An occurrence within an object, for which one or more objects specify interest, is called an **event**. A **notification** is a message, which is sent by an object after an occurrence of an event, in order to notify other objects that specified interest in the event.

The CORBA Event Service consists of three main components. The **supplier** is the sender of a notification. The **consumer** is the receiver of the notification. The **Event Channel** forwards the events from suppliers to consumers. The suppliers and the consumers register for specific events and the channel stores the registration information. The suppliers pass an event that they generated to the Event Channel and the Event Channel delivers the event to the registered consumers. To deliver these events to and from Event Channels, the application developer chooses one of the four following communication models.

a. Communication Models

There are four different communication models in the CORBA 2.2 Event Service Specification: the push-push model, the pull-pull model, the push-pull model, and the pull-push model.

1. In the **Push-Push Model**, suppliers are active. They push the notifications to the Event Channel and the Event Channel pushes notifications of these events to the passive consumers. In this model, the Event Channel plays the role of a notifier.

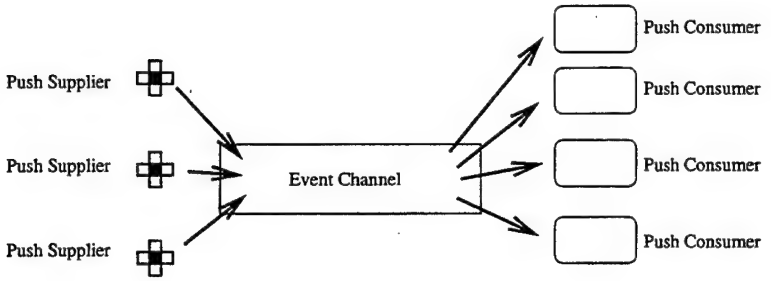


Figure 16. The Push-Push Model

2. In the **Pull-Pull Model**, consumers are the active initiators. They pull the notifications from the Event Channel and the Event Channel pulls the notifications from the suppliers. Since active consumers can get notifications from passive suppliers via the Event Channel, in this paradigm, the Event Channel is called the procurer.

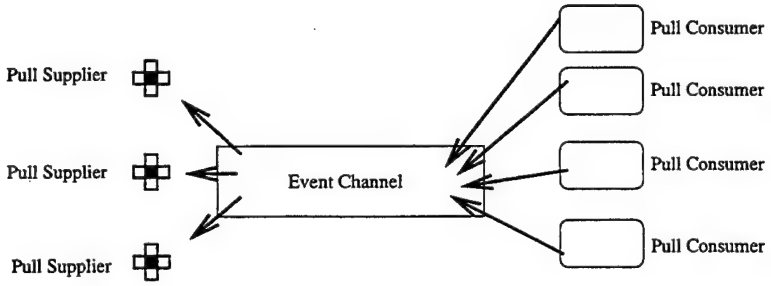


Figure 17. The Pull-Pull Model

3. In the **Push-Pull Model**, both consumers and suppliers are active initiators. The Event Channel plays the role of a queue, where the suppliers push their notifications to the channel and the consumers pull them from the queue.

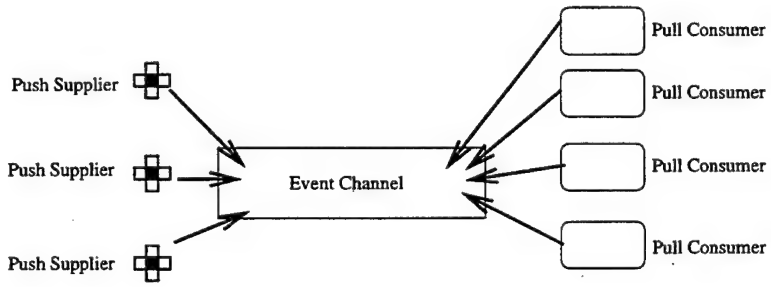


Figure 18. The Push-Pull Model

4. In the **Pull-Push Model**, the Event Channel is the smart agent and the only active component. Both consumers and suppliers are passive. The Event

Channel pulls notifications from the suppliers on behalf of the consumers and pushes the notifications to the consumers.

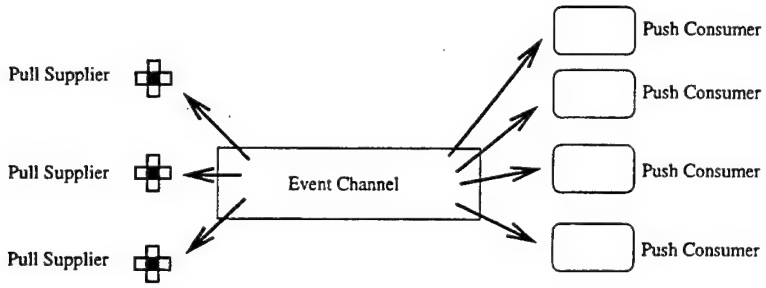


Figure 19. The Pull-Push Model

b. Types of Event Communication

The event communication can be either generic or typed. In the generic case, all requests are passed with generic push or pull operations that use a single parameter, of type any (see Appendix C), to package all of the data passed to the consumer. In the typed event communication, the requests are passed through operations defined in OMG IDL.

1. In **Untyped Event Communication** the push() call takes in a single argument of type any, and the `pull()` call returns a value of type any. If the consumer and supplier do not agree on the actual type, the data received is not useful for the consumer, because it cannot be extracted correctly. The consumer must verify that the `TypeCode`, associated with the returned value, is the type it expected; otherwise errors can occur. The developer of untyped event consumers must provide an implementation for a `push()` or a `pull()` function that filters the notifications and reacts with the expected behavior.
2. In **Typed Event Communication**, a programmer defines an interface that is used by the suppliers and consumers. User defined types may be used as event types. In typed event communication that uses the push model, the supplier may invoke any operation in the intersection of the consumers' interfaces.

2. CORBA Naming Service

The association between a **name** and an object is a **name binding**. A name binding is always defined relative to a **naming context**. A naming context is also

an object that contains a set of name bindings in which each name is unique[1]. The naming context is analogous to a scope in a programming language. An object can be bound to different names in the same or different naming contexts, simultaneously. However, not all objects must have names in order to be available for clients. [Ref. 12]

Binding a name means creating a binding between a name and an object in a given naming context. The reverse operation of binding a name, resolving a name, is to determine the object associated with a name in a given context. There is no absolute name, i.e., a name is always resolved within a given context.

Since a context is also considered an object, it can be bound to a name in a naming context. Binding contexts with names from other contexts creates a directed naming graph with nodes and labeled edges where the nodes are contexts. Such a naming graph allows complex names to reference an object. Given a context in a naming graph, a sequence of names defines a path in the naming graph that can be used to navigate in the resolution process.

The specification of the Naming Service supports distributed, heterogeneous implementation and administration of names and name contexts. Namespaces should be able to be nested and joined at any point in a graph, independent of their implementation. We anticipate that this design will support MSHN's evolving architecture to create our hierarchical structure of components. We discuss this issue further in chapter IV.

D. SUMMARY

In this chapter, we discussed topics from the CORBA 2.2 specification to familiarize the reader with this specification before we discuss our design, our experiences,

[1]The developer cannot use the same name more then once in the same naming context because it will cause ambiguity.

our problems and our solutions. For further information, the reader may refer to the CORBA 2.2 specification [Ref. 11].

IV. DESIGN, LESSONS LEARNED, AND QUANTATIVE RESULTS

Our goal is to determine both (1) how we can best facilitate efficient communication between the components in our architecture using mechanisms from the CORBA 2.2 specification and (2) to determine the run-time overhead of each of those mecahnisms. Our justification for choosing particular mechanisms includes extensibility, scalability, portability, flexibility, and efficiency.

First we describe how the MSHN architecture would benefit from the both the Typed and Untyped Event Service, the Static Invocation Interface (SII), and the Dynamic Invocation Interface (DII). Then we discuss how we use the Naming Service within MSHN to obtain object references. We report problems we experienced and propose solutions. Some solutions recommend, from the user's point of view, improvements to the CORBA specification. Finally we describe the experiments that we designed to measure CORBA overhead and present our results.

A. ALTERNATIVE CORBA DESIGNS FOR COORDINATING MSHN COMPONENTS

MSHN consists of multiple, eventually replicated, distinct distributed components that themselves execute in a heterogeneous environment. These components will have widely varying functionality, will come in and out of existence, will communicate via heterogeneous networks, and will execute on different platforms. To facilitate the interactions between MSHN's components, we identified four mechanisms from the CORBA 2.2 specification for run-time performance examination: both the Typed and the Untyped Event Service, the Static Invocation Interface (SII), and the Dynamic Invocation Interface (DII). After settling on these four mechanisms, we implemented a prototype of MSHN's communication infrastructure using each of them. In this section, we describe each of these designs and our design decisions, the

problems we encountered, and the solutions we arrived at for our architecture.

1. Event Service

Event Service allows multiple suppliers and multiple consumers to deliver and receive notifications for a set of events. An Event Channel transparently permits (1) suppliers to send notifications of events and (2) consumers to receive these notifications without knowledge of the existence of one another. Hence, the Event Service will support transparent replication of MSHN system components for reliability and dependability. Event Service enables Client Libraries, linked with different concurrent applications, to communicate with other system components seamlessly. Finally, Event Service supports a standard Application Programming Interface (API), e.g., for Push-Push model, a single operation `push()`, taking a variable of type `any` (see Appendix C) as a parameter, which eases development of system components.

In the previous chapter we described four models for Event Service. However, when we started the research, there were only two of them available in industrial implementations, the Push-Push Model [Ref. 16] and the Pull-Pull Model.

Because using the Pull-Pull Model creates an additional load on the consumers and because our servers, in this case the consumers, must minimize required computing resources for their functionality, even when there is no event to be delivered on the Event Channel, we chose to use only the Push-Push Model.

a. Using Event Service in MSHN

Figure 20 illustrates the use of Event Service to organize communication in the MSHN architecture. In this approach, the components of MSHN must register themselves as both a consumer and a supplier to the Event Channel. The Event Channel acts as the glue between all of the components and delivers notifications to each of them.

b. Problems with Initial Approach

Although this approach helps to organize MSHN's communication, providing transparent reliability and scalability, some problems with both performance

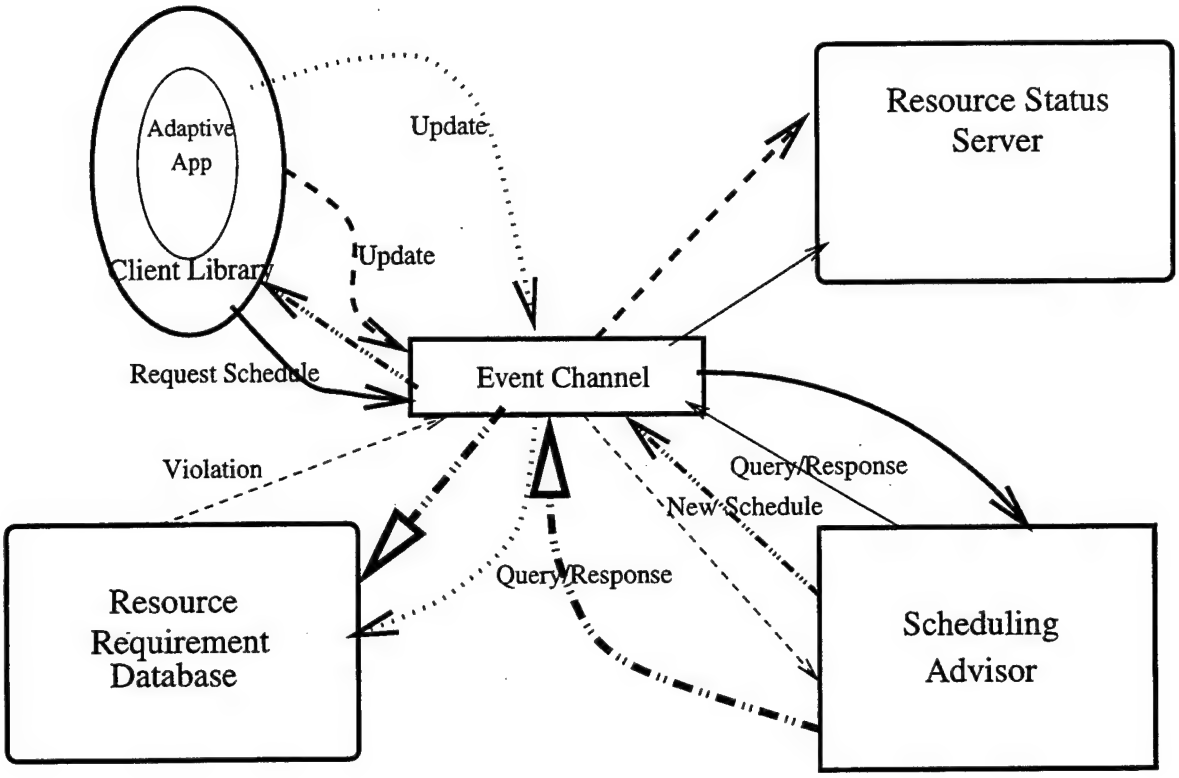


Figure 20. Using Event Service in MSHN

and the CORBA 2.2 specification can be seen. Some of the problems with this approach are identical to the problems identified by Schmidt and Vinoski in the analysis of their stock market application. [Ref. 15] We first summarize their findings in the first two items below, Loss of Events in the System, and Problems with the Untyped Event Service. Then we enumerate additional problems that are particular to using CORBA within the MSHN architecture.

Loss of Events in the System: Event Service guarantees delivery of notifications to all registered consumers as long as the Event Service process does not fail[1]. However, in the Event Service specification, persistency of events in the Event

[1]Although there are many definitions of failure, here we mean that if the Event Service does not fail, then all consumers receive the correct value. This agrees with Lamport's definition of failure:

> If the input unit is nonfaulty, then all nonfaulty processes use the value it provides as input (so they produce the correct output) [Ref. 17].

Channel is not required. Therefore, if an Event Service process does fail, undelivered notifications in the system may be lost. The loss of notifications is fatal for our system because we are creating an environment for mission-critical applications. The obvious solution for this problem is to redefine the Event Service specification to include persistency for the undelivered notifications in the Event Channel. The OMG has been defining this requirement in the Notification Service specification [Ref. 11]. However, no vendors had implemented this new specification at the time of this research.

Problems with Untyped Event Service: The Untyped Event Service does not specify any way to filter notifications. Therefore when using this service all notifications are received by all registered consumers. Passing all of these notifications in MSHN, each of which will be discarded by many consumers, through the network will increase the network load between the Event Channel and the consumer. Additionally, the consumers must filter events and convert the parameters that have type any to the type that is expected. In this case, there is an additional and unwanted load on the consumers to process all the events received. Finally, when more suppliers, in particular more applications, register with the Untyped Event Channel, more events will be generated in the system. Since the Untyped Event Channel delivers each event to all of the registered consumers and the consumers will filter all the events, the network load and consumer load will increase rapidly.

To handle this problem we can use Typed Event Channels, which filter the notifications according to their type. With this solution, the consumers receive only the notifications for which they register, decreasing the network traffic. This solution is shown in Figure 21. In this solution, one Event Channel processes all of the notifications and delivers them only to the corresponding consumers. This also lightens the loads on the consumers because they avoid having to examine and discard events that are not meant for them. However, we note that it increases the computational load on the Event Channel. Later in this chapter we compare the

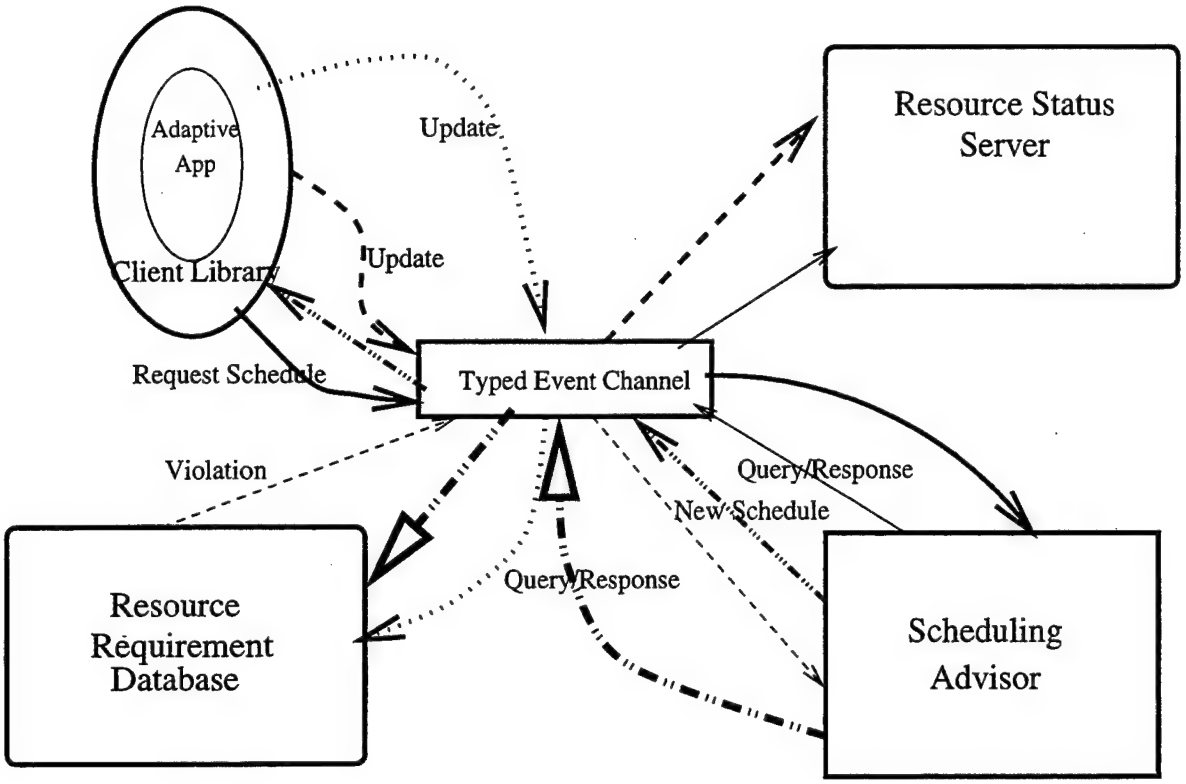


Figure 21. Using Typed Event Service

run-time performance of Typed Event Channel to Untyped Event Channel using this approach in the MSHN architecture.

Alternatively, since we only have five different types of components in MSHN, we may use different channels for each connection between these components. In this approach, each Event Channel will only support one notification type. For example, for the Client Library - Scheduling Advisor Event Channel, we will have the Client Library as a supplier, the Scheduling Advisor as a consumer, and the client schedule requests as the types of the notifications. Each MSHN component may be replicated by registering the additional identical components to the same Event Channel. This solution is shown in Figure 22.

Obviously a combination of these two solutions may be best. That is, in the first solution the Typed Event Channel itself can become a bottleneck. Therefore, replication of Typed Event Channels may better fit the MSHN's requirements. In

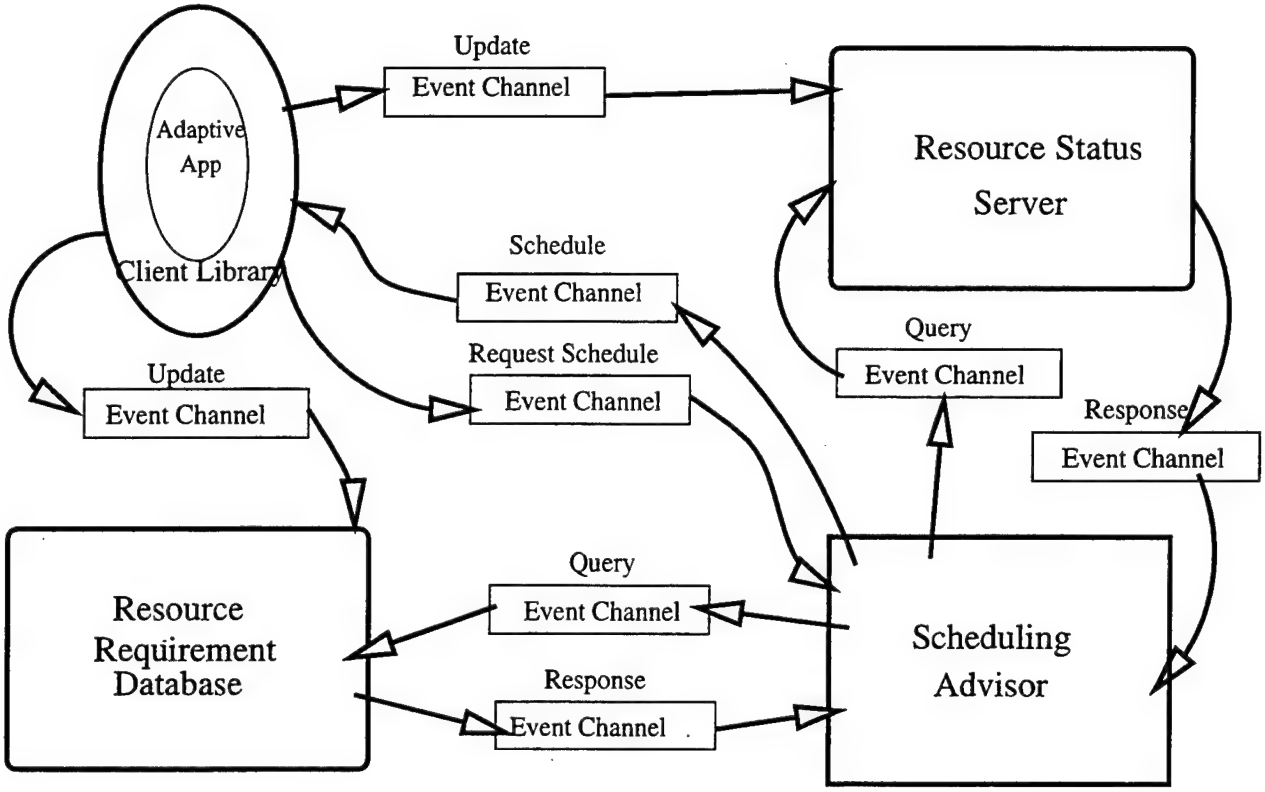


Figure 22. Using UntypedEvent Service

this thesis, we focused on the careful analysis of individual solutions rather than empirically exploring the exponentially sized solution space that combining these two techniques will create.

How to implement a component that is both a supplier and a consumer in a system in order to minimize the run-time overhead: All components of the MSHN are both consumers and suppliers. Also perhaps particular to MSHN, usually when a component receives a notification, it becomes a supplier, generates another notification, and delivers it to the appropriate event channel. Figure 23 shows the process of passing notifications from the Client Library to the Scheduling Advisor (SA) using the `push()` operation, revealing the SA changing from a consumer to a supplier. In the Untyped Event Service Push-Push model, the supplier (here the Client Library) invokes a default `push()` operation on the Event Channel which in turn invokes a `push()` operation supplied by the developer of the

consumer (here the SA). In the `push()` operation that the developer supplied for the SA (as a consumer) the developer of the SA invokes the default push operation on the SA – Resource Requirement Database (RRD) Event Channel (which of course, invokes the `push()` operation supplied by the developer of the RRD). Here the problem is to supply the Interoperable Object Reference (IOR) of the SA – RRD Event Channel to the `push()` operation of the SA. We want to avoid using the Naming Service every time the `push()` operation (here the push operation of the SA) is invoked. Instead the developer can locate the SA–RRD Event Channel in the servant implementation. That is, the servant implementation will obtain IOR for the SA–RRD Event Channel, stringify the IOR, and store it in a file as we discussed in Chapter III. The `push()` operation implementations can retrieve these IORs from the files as needed and deliver generated events, thereby pushing the corresponding notifications to the channel.

Therefore, in the Untyped Event Service, the developer of the consumer (here the Scheduling Advisor) must override the default `push()` operation between the Event Channel and the consumer, to react to the notification (here a request for a schedule) that the consumer receives. For example, when the Scheduling Advisor receives an event from the Client Library requesting a schedule, it will generate a query notification for the Resource Requirement Database and deliver it to the Scheduling Advisor (SA) – Resource Requirement Database (RRD) Event Channel. In this case, the Scheduling Advisor becomes a supplier and requires locating the SA – RRD Event Channel. To avoid locating the Event Channel to which the supplier will deliver the notification, via the Naming Service inside the `push()` operation, the developer can locate the Event Channel in the servant implementation and obtain IORs of it. Then, the servant implementation can stringify these IORs and store them in files as we discussed in chapter III.

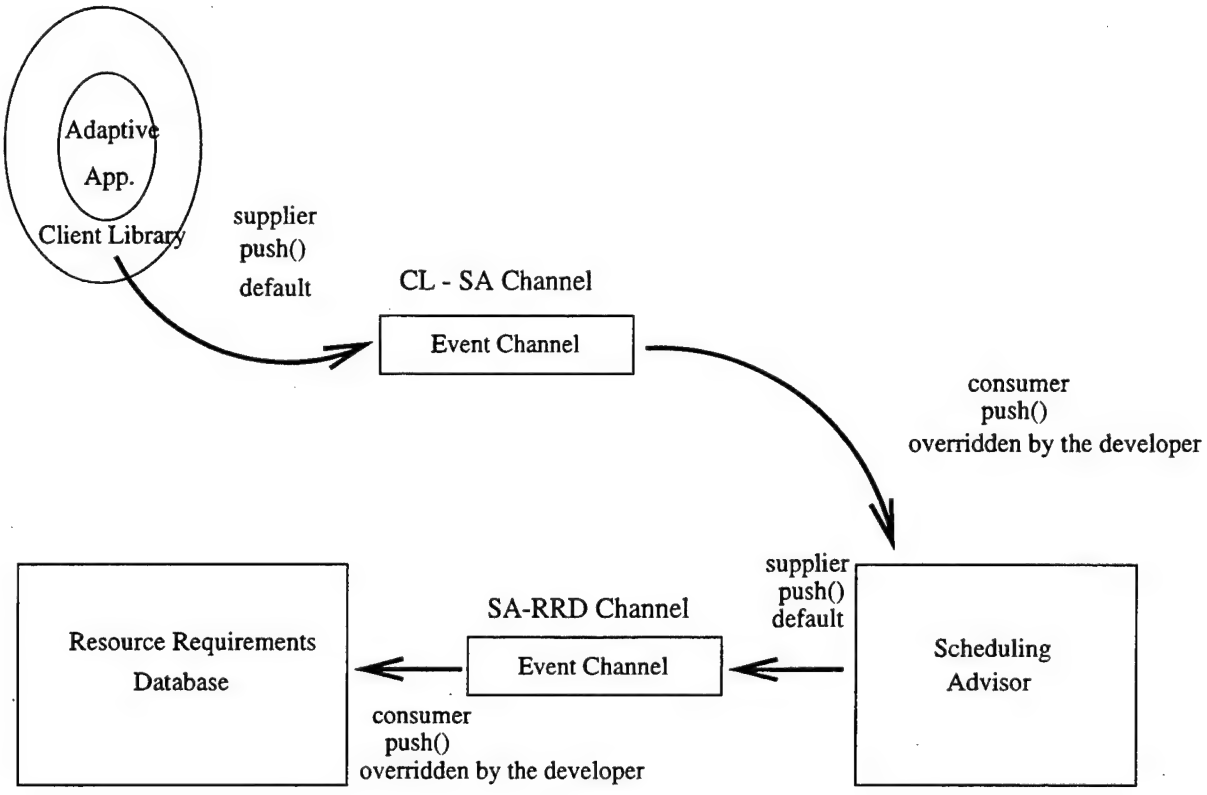


Figure 23. Using push() Operation

2. Remote Invocations

In this section we discuss using remote invocations in MSHN to coordinate the interactions of MSHN components. Since both the Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII) have similar remote invocation mechanisms, first we define the general problems that we encountered with both and then enumerate the additional ones specific to the DII.

Use of remote invocation can provide the same functionality as described above using the Event Service to the MSHN. The most important difference is that the replication of the components is not as easy as it is in Event Service. To support replication using remote invocation, clients must make multiple invocations rather than just one as they would need to in Event Service.

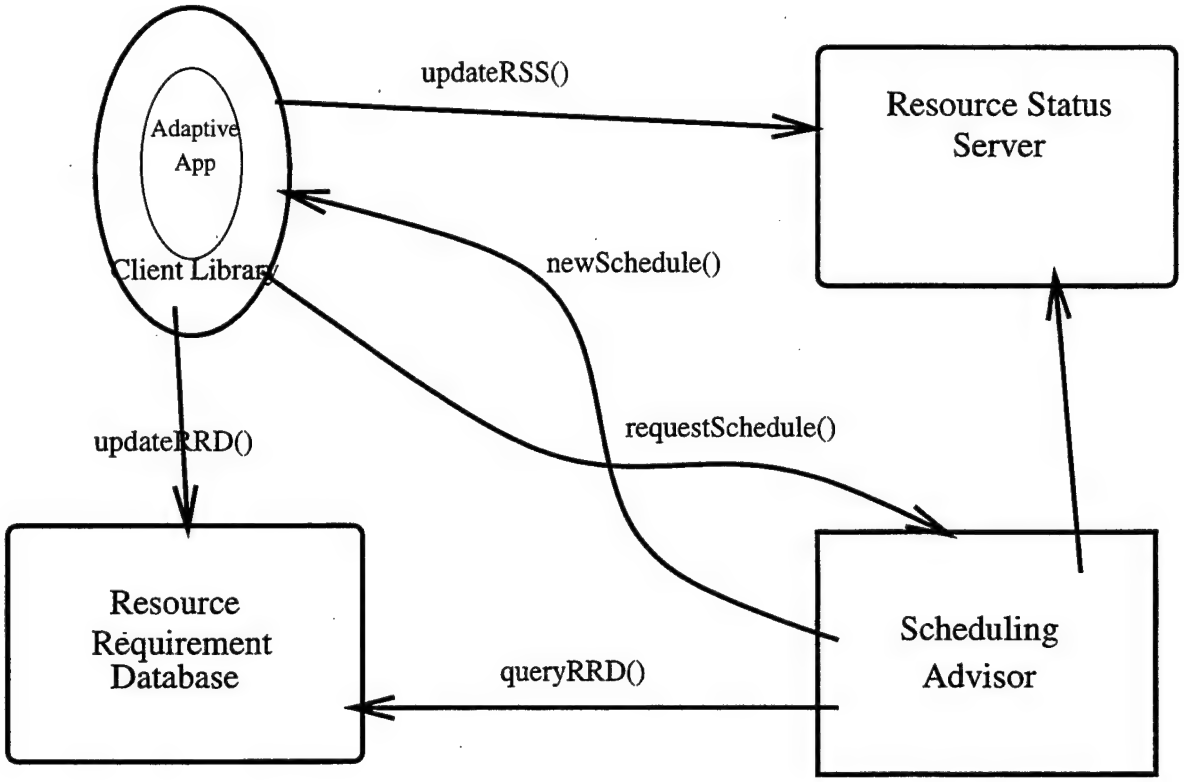


Figure 24. Using Remote Invocations in MSHN

a. General Approach using Remote Invocation

Figure 24 shows our approach that uses remote invocations (i.e., either the Static Invocation Interface (SII) or the Dynamic Invocation Interface (DII)) to establish inter-component communication in the MSHN architecture. We selected from two communication methods available in both SII and DII: the one-way invocation and the synchronous invocation, depending upon whether reliable communication was required.

When using the SII, a component requires compile-time knowledge of the IDL interface of the target component from which it will request a service, whereas the same component, using the Event Service, makes its request via a standard API that is independent of the target component or its functionality. However, when using the DII, the components of the MSHN can invoke operations on other components without requiring precompiled stubs. Thus, we may substitute different instantiations

of system components without requiring a re-linking. Additionally, using the DII will allow us to invoke objects using deferred synchronous invocation. Such invocation is not available from SII within the current CORBA 2.2 specification. With deferred synchronous invocation, the clients may continue their computation instead of waiting for the results of the previously invoked operations to be delivered.

b. Problems with Using the Initial Remote Invocation Approach

We now enumerate some problems with our initial remote invocation approach.

Lack of a Standard Thread Mechanism: Our first design decision was to implement the remote invocations with threads, i.e., handling each invocation at a component using a different thread. Using threads would avoid any data synchronization problems and support fairness for each schedule request. However, the CORBA 2.2 specification does not define how the threads must be implemented. Therefore, each vendor came up with its own solution, which leads to non-portable applications. For example, if you use IONA's Orbix as your development environment, and their Filters to implement your threads, you cannot use the same implementation on Inprise's Visibroker, because their solution for handling threads uses Interceptors.

We avoided the non-compliant extensions of the vendor when we implemented our prototypes. Therefore, we were unable to use threads for any of our prototypes, although the usage of threads would have improved throughput of schedule requests.

Best Effort Semantics: One way invocation has best-effort semantics. Thus, there is no guarantee that the requested method is actually invoked. In this mechanism, the client continues its processing immediately after initializing the request and never synchronizes with the completion of the request. Hence, one way invocation is not a good mechanism for most of the MSHN system because it is not reliable.

However, using one way invocations for frequent short-term updates

could be cost effective in some cases in MSHN. There are two advantages to selectively using best effort asynchronous semantics in the MSHN system between the Client Library and the Resource Status Server. First, the Client Library can continue its computation immediately without blocking. Second, we expect that the Resource Status Server will be updated very frequently. Therefore, we can get the accurate status of a resource with the next update instead of using more reliable transmission mechanisms.

c. Problems with Our Initial Approach that are Specific to using DII

We now enumerate some problems with our initial approach that are specific to using DII.

The Additional Overhead of DII: DII's traditional approach requires 5-6 method invocations to look-up the interface name, get the operation identifier/parameters, and create the request, which may also be remote in order to invoke a single remote method. This adds a lot of overhead to the run-time performance, which would be unacceptable in our MSHN architecture.

In MSHN however, we know the interface of the components, i.e., the operation identifier, the parameters, and the return type when we are developing the client applications. Thus, we can obtain the flexibility and benefits of the deferred synchronous invocation of DII, without having to pay the overhead of querying the Interface Repository for the interface information. We do note that if a deferred synchronous invocation, such as promises [Ref. 18] had been specified as part of CORBA's static invocation interface, the use of DII would not be necessary in this case. We compare the performance of SII and DII in the results section of this chapter.

3. Using the Naming Service

We used the Naming Service to obtain object references in each of our prototypes. For static and dynamic invocation interfaces, all components must resolve names to obtain IORs via the Naming Service only once, when they are instantiated.

References within all components, except the Client Library, are stored in files for future use as we defined in previous section. The components do not use the Naming Service unless the IORs that they have are no longer valid. We will use the exception handling mechanism in CORBA to catch non-valid IORs and to obtain new valid ones by using the Naming Service.

To improve the run-time performance of the Event Service implementations, we registered each component with the appropriate Event Channel. We resolve the Event Channel references using the Naming Service. Then we query the Event Channels to obtain the references for the Proxy Push Suppliers, stringify them, and then store them in files. When a component receives an event, and generates another event in response to the one it received, that component reads the appropriate file to obtain the stringified reference and uses this reference to push the event to the corresponding event channel.

B. QUANTITATIVE RESULTS

We described our design decisions for implementing our prototypes in the previous section. In this section, we discuss the performance results of these different prototypes. First, we describe our test bed. Then we explain our tests and enumerate the results for each of these tests.

1. Hardware and Software Used in the Test Bed

When we began this research we surveyed the available implementations of CORBA to determine what services were supported (See Figure 25). Based upon the robustness and availability of services, particularly the Typed Event Service, we chose IONA Technologies CORBA implementation OrbixMT2.3c, OrbixNames1.1c, OrbixEvent1.0c (Untyped Event Service) and OrbixEvent1.0b (Typed Event Service) that uses the SunSparc C++ Compiler 4.1.

We ran our tests on SunSparc Station 10 hosts with 128 MB of RAM each, and 300Mhz speed CPUs running the Solaris 2.6 operating system. The hosts were

Vendor	*Nm*	*Lf*	*Ev*	*Tr*	*Id*	*Rl*	*Cc*	*Ex*	*Po*	*Tx*	*Sc*
Expersoft	X		X			X					
Sun	X	X	X		X	X					
Iona	X		X	X						X	
Visigenic	X		X							X	
BEA											X
ICL			X							X	X
HP	X	X	X	X						X	
IBM	X	X	X		X		X	X	X	X	
Chorus											
OOT	X			X				X			
Electra	X	X	X								
Xerox											
BBN	X	X							X		

Nm Naming
Lf Life Cycle
Ev Event
Tr Trading
Id Identity
Rl Relationships
Cc Concurrency
Ex Externalization
Po Persistency
Tx Transactions
Sc Security

Figure 25. Available Services

connected through a 100 Mbits/sec Ethernet connection.

To obtain correct results in the tests that utilize the network, we used Network Time Protocol to synchronize the system clocks of the hosts. We found that the system clocks on the SunSparc 10 machines have a skew of approximately 3 milliseconds every 15 minutes. Therefore, we synchronized the clocks every 5 minutes and ran the tests immediately after the synchronization, in order to minimize the difference between the various system clocks.

2. Assumptions Made for Performance Analysis

We determined the overhead of each CORBA mechanism that we described earlier on a single machine. Additionally, we compared reponse times over the network of the various CORBA mechanisms. To compare the overhead, we measured the total time required to service 1000 scheduling requests. This interval begins when the Client

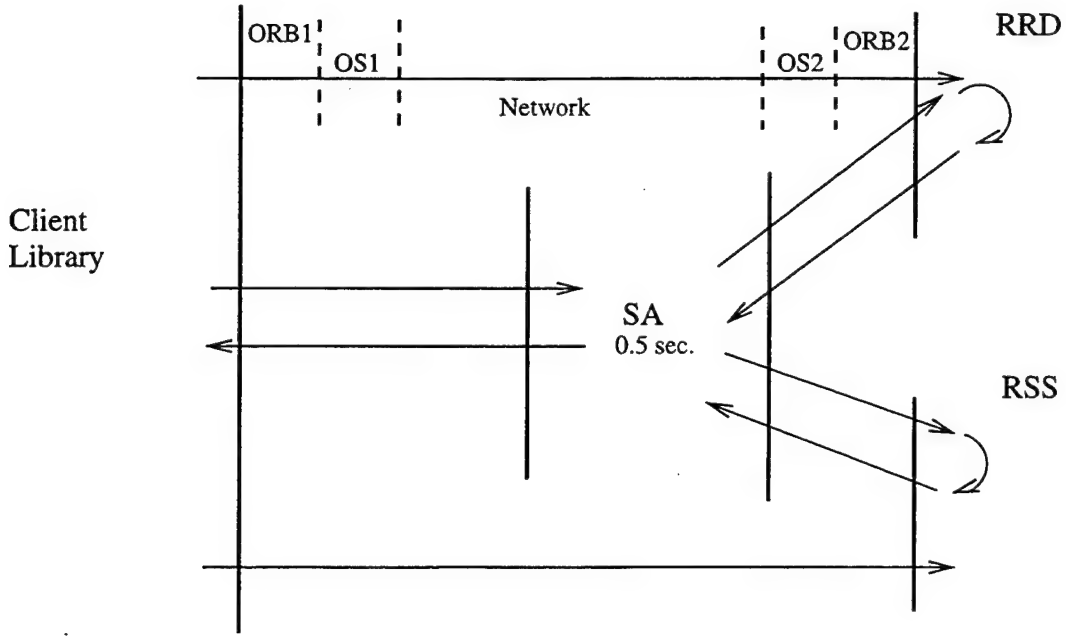


Figure 26. The Emulation of MSHN Communication Infrastructure

Library requests a schedule from the Scheduling Advisor and includes all processing until the time that the Client Library receives a response. This duration includes the time spent querying the Resource Requirement Database (RRD) and the Resource Status Server (RSS). At the time of the testing we did not have a fully functional RRD, RSS and SA. We emulated the SA's execution by having the SA thread that was computing a schedule pause for .5 second(see Figure 26). We chose this duration based upon the average times to execute 11 algorithms proposed by Siegel [Ref. 19] for MSHN's repertoire. Because we expect the RRD and the RSS to be very efficient, we did not force them to pause. In any case, our experiment can easily be repeated by replacing the RRD, the RSS, and the SA with their actual functionality once they are completed.

We ran all tests on a single machine and all tests, except the non-CORBA tests, over the network using four workstations to emulate each of the Client Library, the RSS, the RRD, and the SA respectively. All single machine CORBA tests were executed using four different processes: one emulated all Client Library requests

and the other three emulated the RSS, the RRD, and the SA, respectively. The non-CORBA single machine tests executed completely in a single process, with all MSHN calls being implemented as ordinary C++ function calls. In implementing both the static invocation and dynamic invocation, we used synchronous semantics. We suggest the use of asynchronous invocation in the future work section of the next chapter. We did not have time to implement a fully optimized version of the MSHN communication using sockets to determine the CORBA overheads when running over the network. However, for a class project, Schnaidt and this author implemented another system which we report in a technical report [Ref. 20]. In the following paragraphs, we draw some conclusions based both on these experiments and those reported there.

3. Test Results and Remarks

In this section, we discuss the performance results of the four different prototypes that we described in previous section.

To assess the overhead of CORBA, we included one non-CORBA test. This base case consists of an application linked with all MSHN components and executing as a single process on a single host. This non-CORBA test uses local method invocation to achieve MSHN component intercommunication. We can compare the test cases that use CORBA mechanisms to interconnect MSHN components, all of which are running on the same host as the applications, to this base case in order to assess CORBA's overhead. We then compare these tests against ones where the MSHN components are distributed across different machines.

The average interarrival rate of scheduling requests will vary with the installation and time of day. Therefore we ran all of our tests for two different circumstances. In the first situation, the interarrival rate of the requests is less than the service time, i.e., each request is completed by the system before the next request arrives. The second case represents intermittent bursty loads. In this case, the interarrival rate of the requests is greater than the service time, i.e., some requests must be queued to be

handled later. The first case is important in determining performance under normal conditions, but it is equally important for us to determine that the system neither (1) fails completely when heavily loaded, nor (2) incurs overhead that is exponential in the number of requests pending. Indeed, we did encounter implementations of typed event service, which is very new to the industry, that could not pass our stress test. However, in saying this we must also express our gratitude to IONA for letting us use a Beta version of their software, when no one else even had any version that we could use.

We wanted to initially determine the amount of time required for (1) an application to make a request of the Scheduling Advisor, (2) the Scheduling Advisor to query the RRD to determine what resources are required, (3) the Scheduling Advisor to query the RSS to find out the status of these resources, (4) the Scheduling Advisor to compute a schedule and to return it to the application, and (5) the application to execute and update the RSS and RRD, once each. We first put timers around the five steps and saved the intervals for many applications. Unfortunately, we found that, for our base case, the granularity of the clock was insufficient to accurately measure the difference between the steps above and the total time to compute a schedule and execute the application. We, therefore, modified our normal load case so that we set a timer, generated a request for a schedule, awaited the schedule, then generated the next request. In this way, we generated 1000 requests, and recorded the total amount of time to process all of them. In this case, we used synchronous calls in both DII [2]and SII implementations because we had to await a schedule before submitting the next request. For both Typed and Untyped Event Service implementations, we forced the application to wait for the response of the previous request before starting a new one even though such programming style is counter to the intentions of the designers of the Event Service. Although it is reasonable for all other calls to be synchronous, updates to the RRD and RSS would always be asynchronous. Because we were not

[2]We used the function `invoke()` for DII.

using a multi-threaded base case, we could not make these calls asynchronous in the base case. In order to compare similar situations in local method invocation, SII, DII, and Event Service, we chose to delete these calls from this set of tests. Because requests are generated consecutively, and because each request will use synchronous semantics to make the invocations we call this set of tests **consecutive synchronous** tests.

To simulate the case where lots of requests occur within a short time frame, in our base case, we generated interrupts every .06 seconds. Our interrupt handler submitted a new request to the Scheduling Advisor. For this set of tests, we used asynchronous calls within the application to start the schedule request chain in the DII and SII implementations [3]. Since asynchronous communication is the intended behaviour for the Event Service, in this test case, the application simply executed the `push()` function within the Client Library for the Event Service implementations. From now on we will call this set of tests the **bursty asynchronous** tests because the requests will arrive faster than the expected required service time and they will queue up for the Scheduling Advisor. In this case, we included the update RRD and RSS calls for SII, DII, and the Event Service implementation, even though we did not for consecutive synchronous tests.

We summarize our quantitative results in Figure 27. The times shown are the actual execution times in seconds for 1000 requests. We have included a scheduling time of .5 seconds per request and have not simulated the execution time of the application.

Before we analyse our results, we would like to discuss two Unix system calls that we used to simulate the .5 seconds scheduling time as well as the bursty request arrivals. We used the `select()` system call's timeout parameter to emulate the time required for the Scheduling Advisor to compute a schedule in all except the bursty asynchronous base case. We initially also used the `select()` system call in the base

[3] We used the function `send_oneway()` for DII, and one-way semantics for SII.

Configuration	Communciation Mechanism	Local for 1000 Requests	Network for 1000 Requests
Consecutive Synch.	Non-CORBA	500.1	N/A
	SII	511.4	520.0
	DII	530.1	530.4
	Untyped Event	607.4	593.9
	Typed Event	580.5	779.2
Bursty Asynch.	Non-CORBA	500.1	N/A
	SII	510.8	510.8
	DII	521.2	520.2
	Untyped Event	592.8	564.4
	Typed Event (for 100 requests)	64.7	63.6

Figure 27. Results of the Generic Experiments

case as well, but ran into problems when compiling that call with `ualarm()` system call that we used to generate the next request in the bursty asynchronous tests. After the `ualarm()` signal handler completed, control which had resided in the `select()` call returned instead to the statement after it, causing the emulated schedule to prematurely complete. The average of the values we obtained from calling `select()` was 125 microseconds above the timeout value of .5 seconds. As mentioned above, we used the system call `ualarm()` to give asynchronous behaviour to the C++ function call implementation and observed an average 10 milliseconds error for a set value of 60 milliseconds. Both of these biases are beneath the typical Unix system call granularity of 10 milliseconds.

We note that, of course, there is significant overhead in using CORBA between address spaces as compared to local method invocations within a single address space. We note also that a similar result was shown in our technical report between CORBA and local Inter Process Communication (IPC) [Ref. 20]. We conclude in that report that the efficiency of the socket implementation on one machine is due to use of shared memory in that socket implementation. We also note that even if the CORBA implementations were to use shared memory, when available, that similar performance

Communication Mechanism	Added Overhead
SII	10.5
DII	20.0
Untyped Event	64.2
Typed Event	13.5

Figure 28. Added Overhead for Bursty Asynchronous Test Case over the Network

enhancement would not be obtained. This is because the CORBA specification requires all parameters of the request to be converted to External Data Representation (XDR) and then sent to the target object no matter where the target object resides. Also, in that report we noted that a CORBA implementation, which required less than 5% of the time to implement as compared to the socket implementation, had only 20% more run-time overhead. Since our results are comparable here, and because we did not have sufficient time to implement a highly optimized MSHN socket implementation, we will limit the remainder of our remarks to comparing the performance of various CORBA implementations.

Since local invocation is done within the same address space, the performance of the local invocation as compared with any of other prototypes is substantially faster than the CORBA implementations, as would be expected. The Static Invocation Interface is generally the fastest mechanism available in the CORBA specification [Ref. 5]. Even though the Dynamic Invocation Interface is generally considered much slower compared to the SII, the performance of the DII given our environment, i.e., at component development time we know the interface of the component, is close to the SII performance. Therefore a developer need not necessarily fear the overhead of DII as observed by Orfali et.al. [Ref. 5], if their situation is similar to ours. However, we note that it would probably be more efficient if CORBA makes deferred synchronous semantics available in SII. Therefore, a developer can use this approach especially if deferred synchronous calls, which are at this time specified for DII, are more appropriate for the application.

The comparison between the consecutive synchronous and bursty asynchronous

CORBA implementations seems surprising at first glance. One would normally expect that a system loaded with bursty requests would not perform better than an unloaded system. To understand the reason for this performance improvement, we must describe more detail about our CORBA implementation than we previously gave. In particular, we now further elaborate on the client application's use of Naming Service. In the consecutive case, the Client Library obtains the IOR of the Scheduling Advisor from the Naming Service immediately prior to making each request. However, in the bursty asynchronous case the Client Library obtains all the IORs asynchronously in the first 60 seconds of the actual run-time. Thus in the bursty asynchronous case obtaining the IORs overlaps the actual computation. Unfortunately, in the actual MSHN implementation, unless it is executing on a dual processor machine we would not expect to see this. What is happening is that the emulated Scheduling Advisor is actually blocked while the Naming Service is resolving addresses. In an actual implementation, both would require the use of a CPU. We also observe that there may be another source of this speedup: the Client Library process can send more than one request within its use of CPU, which results in fewer context switches.

In the 4-machine, networked tests we did not need an excessive number of context switches between our components and object request broker. Multiple processes could actually execute simultaneously, and actual run times were smaller. Even though we could not get paralellism in our distributed communication intensive application [Ref. 20], we observed speedup in MSHN because the Scheduling Advisor is computation intensive.

As seen in Figure 27, the Untyped Event Service adds more overhead than either the SII or the DII because the Event Service process is the bottleneck in the system. Of course, in an overall evaluation this additional overhead must be balanced against the ease of replication of system components that it provides. We suspect that much of this overhead is caused by the time required to insert parameters into a type any variable for the supplier component and the time that the consumer spends

	Replication Mechanism	All Hosts	SA and Client Hosts	RRD and RSS Hosts
Bursty Asynch.	Two Event Pro.	N/A	574.38	561.44
	Four Event Pro.	560.98	N/A	N/A
Consecutive Synch.	Two Event Pro.	N/A	599.05	593.82
	Four Event Pro.	593.82	N/A	N/A

Figure 29. Results of the Untyped Event Service Special Cases

to extract the values from a type any variable. Therefore, this mechanism may not be a choice for a developer unless it is really required for the application.

In addition to the tests described above, we replicated the Untyped Event Service to see whether any speedup could be obtained by distributing the load of the Event Service process. First we created two Event Service processes one on the same host as the application and the other on the same host as the Scheduling Advisor in an attempt to achieve speed up. This approach performed worse than the single Event Service process. Upon analysis, we determined that it introduced unnecessary network communication and placed the Event Service processes on the busiest hosts. Then we moved the Event Service processes to the same hosts as the RRD and the RSS, respectively. Figure 29 shows the speedup we observed with this configuration. After these results we decided to try four Event Service processes. Unfortunately, probably because of the excessive amount of communication, this approach performed no better than using a single Event Service process.

In MSHN's Typed Event Service implementation, all of the communication is passed through a single process. The implementation that we used could not efficiently handle 1000 requests in a minute. In the bursty asynchronous case, we observed starvation in communication. We believe that starvation occured because the Client Library was updating the RRD and the RSS whereas the Scheduling Advisor was querying these two components to compute the schedules simultaneously. On the other hand, in the consecutive synchronous case, the Typed Event Service process handled 1000 requests probably because we excluded the update calls to the

RRD and the RSS and because the maximum number of requests handled during the peak time was substantially less in this test. Hence, we believe that the Typed Event Service is not ready to be used in the MSHN. In Figure 27 we represent 100 requests for the bursty asynchronous case. Since the Typed Event Service implementation that we were using did not allow us to replicate it, we could not run a replicated test with the Typed Event Service as we did with the Untyped Event Service.

C. SUMMARY

In this chapter, we described how the MSHN architecture would benefit from both the Typed and Untyped Event Service, the Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII). Then, we discussed how we used the Naming Service within MSHN to obtain object references. We also reported problems we experienced and proposed solutions. Some solutions recommended, from the user's point of view, improvements to the CORBA specification. Finally we described the experiments that we designed to measure CORBA overhead and presented our results. The overhead added by CORBA for distributed component communication of MSHN system varied from a low of 10.6 miliseconds/service request to a high of 279.1 miliseconds/service request on UltraSparc10 boxes with Solaris 2.6 Operating System connected via 100 Mbits/sec Ethernet.

V. SUMMARY AND FUTURE WORK

In the Heterogeneous Processing Laboratory at the Naval Postgraduate School, we are designing, implementing, and testing a resource management system called the Management System for Heterogeneous Networks (MSHN). MSHN is part of the Defense Advanced Research Projects Agency (DARPA) sponsored Quorum program. MSHN's goal is to support the execution of multiple, disparate, adaptive applications in a dynamic, distributed heterogeneous environment. To accomplish this goal, MSHN consists of multiple, eventually replicated, distinct distributed components that themselves execute in a heterogeneous environment. These components will have widely varying functionality, will come in and out of existence, will communicate via heterogeneous networks, and will execute on different platforms. These system components are also likely to be written in different programming languages. We can, of course, at the expense of a great deal of programmer's time, implement from scratch, specialized naming services to locate the appropriate component at run-time and specialized communication mechanisms to enable communication between these heterogeneous platforms. Alternatively, we can use a general tool, such as Common Object Request Broker Architecture (CORBA), to achieve the same functionality while reducing the development time. Experience with generalized systems such as CORBA, has revealed that the reduced development time cost comes at the expense of run-time performance, which can be critical, in real-time applications. This thesis, therefore, investigates the utility and overhead of communication mechanisms from the CORBA 2.2 specification to support MSHN's inter-component communication. In the second chapter of this thesis we provided a brief Client/Server approach for the MSHN architecture. For further information about the MSHN, the reader may refer to a technical report describing the entire project [Ref. 4].

CORBA is an evolving, open standard, which is being defined by the Object Management Group (OMG) to bring some order to the rapid and disjoint development

of object technologies. The OMG is a coalition of over 900 companies, some of which are system developers, and others are users. The OMG's main objective is to influence the object technologies. They define the Object Management Architecture (OMA) Reference Model, upon which all OMG specifications are based. CORBA, the most commonly used OMG specification, supports the construction and integration of object-oriented software components in heterogeneous distributed environments.

In the third chapter of this thesis, we provided a brief overview of the CORBA 2.2 specification to give a basic understanding to the reader about these two concepts. This overview includes definitions of some elements from the OMA Reference Model to familiarize the reader with the terminology. Additionally, it describes two method invocation mechanisms from the specification, as well as two CORBA services, the Naming Service and the Event Service, that we used in our prototypes. For further information, the reader may refer to OMG documentation and other references listed in the bibliography of this thesis [Ref. 5, 6, 7, 8, 9, 10, 21, 11, 13, 12].

Our goal is to determine both how we can best facilitate efficient communication between the components in our architecture using mechanisms from the CORBA 2.2 specification that were briefly discussed in the third chapter as well as to determine the run-time overhead of each of those mecahnisms. To build MSHN's communication infrastructure, we identified four mechanisms from the CORBA 2.2 specification for run-time performance examination: the Static Invocation Interface (SII), the Dynamic Invocation Interface (DII), Typed Event Service and Untyped Event Service. Our justification for choosing particular mechanisms includes extensibility, scalability, portability, and flexibility in a cost-effective way. After settling on these four mechanisms, we implemented a prototype of MSHN's communication infrastructure using each of them, and measured their respective run-time overhead. In the fourth chapter, we provide a rationale for our design decisions concerning the use of CORBA for implementing the prototype of the MSHN Client/Server architecture. In this chapter we describe how the MSHN architecture would benefit from

both the Typed and Untyped Event Service, the Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII). Then, we discuss how we use the Naming Service within MSHN to obtain object references. We report problems we experienced and propose solutions. Some solutions recommend, from the user's point of view, improvements to the CORBA specification. Finally we describe the experiments that we designed to measure CORBA overhead and present our results. The overhead added by CORBA for distributed component communication of MSHN system varied from a low of 10.6 milliseconds per service request to a high of 279.1 milliseconds per service request on UltraSparc10 machines with Solaris 2.6 Operating System connected via 100 Mbits/sec Ethernet.

A. FUTURE WORK

In this thesis, we focused on the run-time performance of different prototypes implemented using four different communication mechanisms from the CORBA 2.2 specification. Since MSHN requires a cost-effective communication infrastructure, i.e., a fast but also a reliable communication infrastructure, we recommend a further examination of these different communication mechanisms with regard to reliability because the fastest communication mechanism may not be reliable enough for MSHN.

We anticipate a hierarchical structure for the MSHN components similar to the Domain Name Service system of the Internet. Further investigation is needed to determine how to use the Naming Service to organize MSHN's component structure to locate the hierarchical components in a sparce WAN.

When locating an object in the system using the current ORBs and Naming Service, an application obtains an object reference to the first available object implementation. This random choice may result in poor service. We expect that research results from Management System for Heterogeneous Networks will influence the evolving CORBA specification so that the specification will include smarter request brokers and smarter trading services.

APPENDIX A. ABBREVIATIONS

BOA - Basic Object Adapter

COM - Component Object Model

CORBA - Common Object Request Broker Architecture

DoD - Department of Defense

DII - Dynamic Invocation Interface

DSI - Dynamic Skeleton Interface

GIOP - General Inter-ORB Protocol

IDL - Interface Definition Language

IIOP - Internet Inter ORB Protocol

IOR - Interoperable Object Reference

IR - Interface Repository

MSHN - Management System for Heterogeneous Systems

OMA - Object Management Architecture

OMG - Object Management Group

OODBMS - Object Oriented Database Management Systems

ORB - Object Request Broker

RPC - Remote Procedure Call

RSS - Resource Requirement Database

RSS - Resource Status Server

SA - Scheduling Advisor

TCP/IP - Transmission Control Protocol / Internet Protocol

APPENDIX B. COMPONENTS OF MSHN ARCHITECTURE

We gratefully acknowledge Mathew Schnaidt [Ref. 22] for letting us use his descriptions of the MSHN components verbatim in this appendix. They are included for the convenience of the reader.

1. CLIENT LIBRARY

The client library is linked with both adaptive and non-adaptive applications. It provides a transparent interface to all of the MSHN services [Ref. 1]. The client library performs at least the following functions: (1) it intercepts system calls to record resource requirements; (2) it forwards requests to start another process, when appropriate, to the Scheduling Advisor; and (3) it intercepts and performs the appropriate action on requests from the Scheduling Advisor to adapt. It forwards the recorded resource requirements to the Resource Requirements Database. The final implementation of MSHN will be able to forward the performance measurements and resource requirements through the MSHN daemon when that is more efficient.

2. SCHEDULING ADVISOR

The Scheduling Advisor performs the highly complex task of scheduling multiple jobs, from multiple users, onto one (or several) computers from a pool of heterogeneous computing platforms. The sophisticated algorithms that the Scheduling Advisor will use to make decisions are beyond the scope of this thesis. However, this research requires knowledge of the interfaces presented by the Scheduling Advisor. The Scheduling Advisor will accept scheduling requests from the client libraries. The Scheduling Advisor will query both the Resource Status Server and the Resource Requirements Database. These queries must respond with near real-time information on the status (load) of the VHM, and the resource requirements of the application. Once the Scheduling Advisor receives this load information, it can calculate a mixture of computing and network resources that will, with high probability, deliver the requested quality of service.

Additionally, in case of a significant deviation from the initial resource status estimate, the Scheduling Advisor will receive notification from the Resource Status Server. For example, if a communications path is severed, or a machine fails, the Scheduling Advisor will be notified and can recalculate a

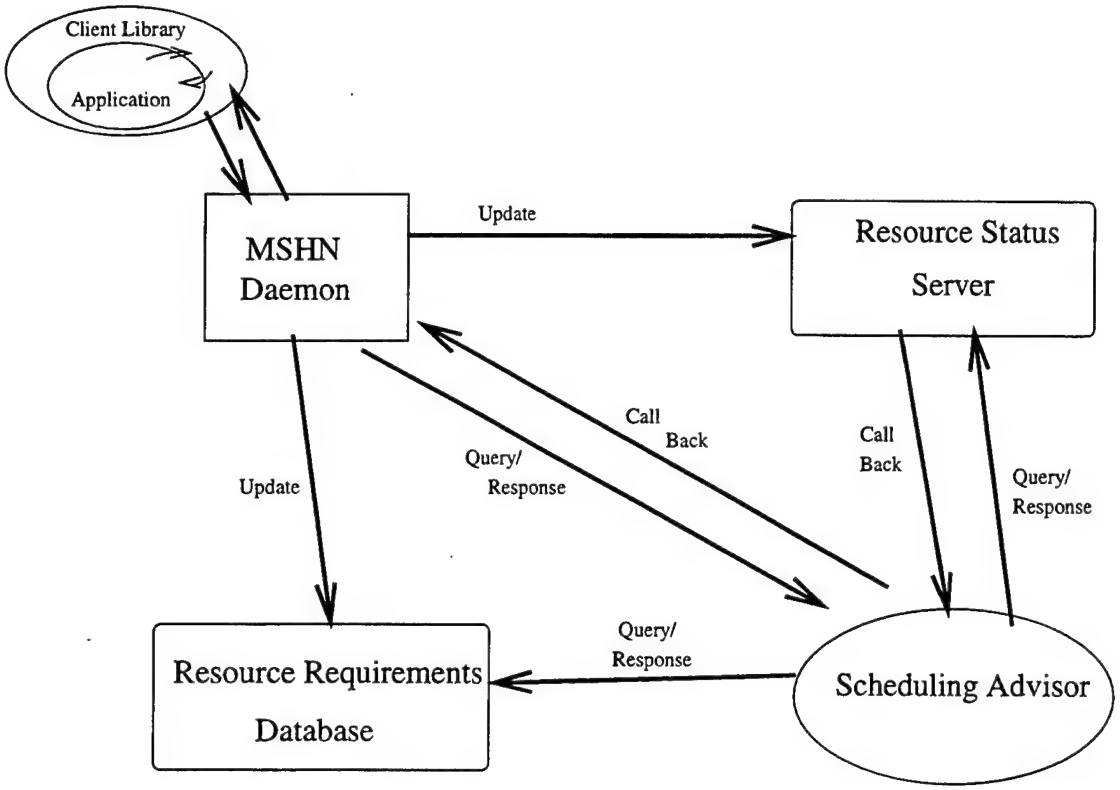


Figure 30. MSHN's Software Architecture

new scheduling solution for the affected applications. The Scheduling Advisor may then signal the client library and advise it that the application should begin using a different algorithm, or perhaps recommend that it shift execution to a different set of resources.

3. RESOURCE REQUIREMENTS DATABASE

The Resource Requirements Database is a repository of information pertaining to the execution of user applications. A job consists of the code and data required to execute a user's application. This database contains statistics on the run-time characteristics of jobs, such as CPU, memory, and disk usage. The Resource Requirements Database provides this information to the Scheduling Advisor upon request. The client libraries update it.

4. RESOURCE STATUS SERVER

The purpose of the Resource Status Server is to maintain a repository of the three types of information about the resources available for MSHN to schedule: the relatively static (e.g. CPU speed), the moderately dynamic (e.g.,

operating system version), and the highly dynamic information (e.g., network load). The Scheduling Advisor will query the Resource Status Server to obtain an initial estimate of the currently available computing and networking resources. After making a scheduling decision, the Scheduling Advisor will notify the Resource Status Server of the additional loads that it expects the client application to place on the compute and networking resources. Much of this thesis is dedicated to determining the best mechanisms for obtaining this most dynamically changing type of information for network resources. [SCHN98]

Periodically during the execution of an application, the client library will update the Resource Status Server with status of the computing and networking resources in use by the application. Also, as described in subsection 2 above, the Resource Status Server sets up a callback with the Scheduling Advisor. If the perceived loads on the resources fall outside a specified range, the Resource Status Server will notify the Scheduling Advisor.

5. THE MSHN DAEMON

The MSHN Daemon executes on all compute resources available for scheduling by the MSHN Scheduling Advisor. It is used to begin and control the execution of processes that are submitted to MSHN.

APPENDIX C. INTEROPERABILITY IN CORBA 2.2

In this appendix we define two of the components of CORBA in more detail that are responsible for interoperability. The Interface Definition Language is used to declare interfaces in a manner that is independent of platform and programming language. The General Inter-ORB Protocol supports the simultaneous use of CORBA implementations from multiple different vendors. This functionality is required to support availability, simplicity, scalability, generality, and architectural neutrality.

1. THE INTERFACE DEFINITION LANGUAGE (IDL)

IDL is a descriptive language that has the same lexical rules as C++. In addition, new keywords are introduced to support distributed systems related concepts. IDL allows the developer to define new, more complex interfaces by inheriting from existing ones.

a. The Structure of CORBA IDL

CORBA IDL consists of several elements. Modules provide a namespace in which to group a set of interfaces (see Figure 31). If the developer is using a C++ to IDL mapping, depending upon the compiler, the modules may be mapped to C++ namespaces or C++ classes. Interfaces and operations were discussed in section A of chapter III. Hence, we do not elaborate on those any further here. We will now discuss data types supported by IDL.

b. CORBA IDL Types

As we have seen in the previous section, OMG IDL provides declarations similar to those of the C programming language that can be used to associate an identifier with a type. OMG IDL uses the typedef keyword to associate a name with a data type; a name is also associated with a data type via the struct, union, and

```
module <identifier>                              Defines a naming context
{
<type declarations>
<constant declarations>
<exception declarations>

interface<identifier> [:<inheritance>]           Defines a CORBA class
{
   <type declarations>
   <constant declarations>
   <exception declarations>
   [<op\_type]<identifier>(<parameters>)         Defines a method
      [raises exception][context];

      . . .

   [<op\_type]<identifier>(<parameters>)         Defines a method
      [raises exception][context];
}

interface<identifier> [:<inheritance>]           Defines a CORBA class
{
   . . . .
}
}
```

Figure 31. The Structure of an IDL File

enum declarations. The data types supported by OMG IDL are classified as (1) basic types, (2) constructed types, and (3) template types.

i. Basic Types

The basic types, except type any, are similar to the basic types one can find in any programming language. The supported basic types are

- long 32 bit arithmetic types (signed and unsigned),
- short 16 bit arithmetic types (signed and unsigned),
- IEEE 754-1985 floating point types,
- IEEE 754-1985 double point types,
- IEEE 754-1985 long double point types,
- character,
- wide character,
- Boolean,
- octet 8-bit value,
- any.

The type any is a tagged type specific to CORBA that can hold any built-in or user defined types. When the type of the parameter to an operation cannot be determined at compile-time, a parameter of type any should be used. At run-time, a value of any type, including user defined types, can be inserted into a variable of type any and passed to an operation. The target object that receives the parameter of type any can examine the parameter's `TypeCode`, and determine the actual type passed to it.

ii. Constructed Types

The constructed types that are supported by OMG IDL are as follows:

- Struct - a struct is an aggregate data type that is built using other data types. (similar to C/C++ `structs`)
- Discriminated Union - OMG IDL unions must be discriminated. The union header must specify a typed tag field. An OMG IDL union value is constructed with a type discriminator and a value chosen from a set of possible types specified in a union definition.
- Enumeration - an enumerated value is chosen from an ordered list of identifiers. `Typedef` declarations can also be used to specify an enumerated type[Ref. 11].

iii. Template types

The following template types are supported by OMG IDL.

- String - a string is similar to a sequence of characters. If a maximum size is specified in a declaration, the string is bounded. Otherwise, it is unbounded.
- Sequence - a sequence is a one-dimensional array with a compile-time defined maximum size and a run-time instantiated length.

2. THE GENERAL INTER-ORB PROTOCOL (GIOP)

The General Inter-ORB Protocol (GIOP) defines ORB interoperability standards. GIOP can be mapped on any connection-oriented transport protocol. A specific mapping of GIOP, which runs over TCP/IP connections, called the Internet Inter-ORB Protocol (IIOP) is also defined in this section. CORBA specification 2.2 states several goals of General Inter ORB Protocol:

- Widest possible availability,
- Simplicity,
- Scalability,
- Low cost,
- Generality,
- Architectural neutrality.

To accomplish these goals, the GIOP specification consists of several different elements. These elements are

- the Common Data Representation (CDR),
- the GIOP message formats,
- the GIOP transfer assumptions, and
- the Internet Inter-ORB Message Transport.

The following paragraphs provide an abstract description of each of the elements.

a. The Common Data Representation (CDR)

A fundamental problem in interoperability is to overcome the different data representations, including byte orders of different platforms, and different languages. The Common Data Representation (CDR) addresses this problem by defining how to represent basic types and `TypeCode`'s independent of platform and language.

b. GIOP Message Formats

The GIOP supports all of the functions and behaviors of the CORBA specification including dynamic object location with several and simple message formats.

c. GIOP Transfer Syntax

The GIOP defines connection management, request multiplexing, and connection usage over any connection-oriented transfer protocol. In particular, GIOP specifies how a connection between a client and server ORB may be shared by different requests while dictating message ordering rules during these transfers.

d. Internet Inter-ORB Protocol (IIOP)

The IIOP specification describes the transfer syntax that must be used when GIOP is implemented on top of TCP/IP, that is, how ORBs open TCP/IP connections and use them to transfer GIOP messages such as those between the client's and server's ORBs [Ref. 11]. Since IIOP runs directly on top of TCP/IP, it is platform independent. Objects are identified and located through the Interoperable Object References, which are defined in the CORBA 2.2 specification. Each IOR has an IIOP profile that contains the Internet address and port number of the object's server and a key value used by the server to find the specific object described by the reference.

APPENDIX D. A SAMPLE INTERFACE

In this appendix, we introduce the reader with a sample IDL file based on our Dynamic Invocation Interface (DII) prototype.

Notice that in the IDL file in Figure 32, there is a specific keyword oneway which indicates that the client will run concurrently with the object when this method is invoked. The developer of the client need not be aware that this is the case because one-way static invocation is only permitted with functions that have no input/output parameters, no output parameters, and no return value. In contrast, Dynamic Invocation clients must be aware of which type of invocation they are making depending on the signature of the operation.

```
//*****************************************************************
// This file includes the interface of the Resource Status Server.
// The type definitions are in mshn_common.idl file.
//
//*****************************************************************
#ifndef _RSS_IDL_
#define _RSS_IDL_

// start with the type definitions
#include "mshn_common.idl"

interface RSS {

        oneway void updateRSS (in RSS_St updateRStat);
RSS_St queryRSS (in RSSReq_St req);
};
#endif
```

Figure 32. A Sample IDL File

```
//*****************************************************************
// This file includes the type definitions used in MSHN's prototypes.
//
//
//*****************************************************************
#ifndef _MSHN_COMMON_IDL_
#define _MSHN_COMMON_IDL_

// start with the type definitions
typedef long memSizeType;
typedef long cpuSpeedType;
typedef long discSpaceType;
typedef unsigned long IPadressType;
typedef long hostIDType;
typedef long netBWType;
typedef long netLatencyType;
typedef long processNameType;
typedef long noOfAccessType;
typedef long sizeOfAccessType;

struct RSS_St
{
        hostIDType      hostID;
        cpuSpeedType    cpuLoad;
        memSizeType     memLoad;
        cpuSpeedType    cpuLoad;
        memSizeType     memLoad;
        discSpaceType   discLoad;
        netBWType       netBWLoad;
        netLatencyType  netLatency;
        netLatencyType  networkLatency;
        memSizeType     cacheLoad;
};
```

Figure 33. The Type Definitions Used in Prototypes

```
struct RRD_St
{
        processNameType prName;
        noOfAccessType  noOfLocalAccess;
        noOfAccessType  noOfRemoteAccess;
        sizeOfAccessType  sizeOfLocalAccess;
        sizeOfAccessType  sizeOfRemoteAccess;
        memSizeType minMemReq;
        netLatencyType networkLatency;
        memSizeType    cacheLoad;
};

struct ClientReq_St
{
        processNameType prName;
        hostIDType hostID;
        string SA_ref;
};

struct RRDReq_St
{
        processNameType prName;
        string SA_ref;
};

struct RSSReq_St {
        hostIDType hostID;
        string SA_ref;
};
```

Figure 34. The Type Definitions Used in Prototypes, continued

```
struct ClientResp_St {
        hostIDType hostID;
        hostIDType targetHostID;
        hostIDType hostID;
        hostIDType targetHostID;
        processNameType prName;
        string RRD_ref;
        string RSS_ref;
};

#endif
```

Figure 35. The Type Definitions Used in Prototypes, continued

LIST OF REFERENCES

[1] John Kresho. Quality network load information improves performance of adaptive applications. Master's thesis, Naval Postgraduate School, September 1997.

[2] Stephen G. Marks and William F. Samuelson. *Managerial Economics.* The Dryden Press, Orlando, 1995.

[3] Alex Berson. *Client/Server Architecture.* McGraw Hill, 1996.

[4] Debra Hensgen and Taylor Kidd. MSHN, in preparation. October 1998.

[5] Jeri Edwards Robert Orfali, Dan Harkey. *Instant CORBA.* John Wiley, New York, 1997.

[6] Jeri Edwards Robert Orfali, Dan Harkey. *Distributed Objects.* John Wiley, New York, 1997.

[7] Sean Baker. *CORBA Distributed Objects Using Orbix.* Addison Wesley Longman Limited, Essex, 1997.

[8] Jeri Edwards Robert Orfali, Dan Harkey. *The Essential Client/Server Survival Guide.* John Wiley, New York, 1997.

[9] IONA Technologies PLC. *Orbix Programmer's Reference Manual*, October 1997.

[10] IONA Technologies PLC. *Orbix Programmer's Guide*, October 1997.

[11] Object Management Group. *CORBA 2.2 Specification*, February 1998.

[12] Object Management Group. *Naming Service Specification*, November 1997.

[13] Object Management Group. *Event Service Specification*, November 1997.

[14] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), February 1997.

[15] Douglas C. Schmidt and Steve Vinoski. Overcoming drawbacks in the OMG events service (column 10). *SIGS C++ Report Magazine*, June 1997.

[16] IONA Technologies PLC. *OrbixEvents Programmer's Guide*, December 1997.

[17] Marshall Pease Leslie Lamport, Robert Shostak. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[18] B. Liskov and L Shirira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *Proceedings SIGPLAN'88 Conference Programming Design and Implementation*, 1988.

[19] Tracy D. Braun et.al. A taxomony for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. *Proceedings of the Workshop on Advances in Parallel and Distributed Systems (ADAPS)*, 1998.

[20] Matthew Schnaidt and Alpay Duman. A comparison of Unix sockets and CORBA in a distributed communication intensive application. Technical Report 3, Naval Postgraduate School, 1998.

[21] Ted G. Lewis. Where is client/server software headed? *IEEE Computer Magazine*, pages 49–55, April 1995.

[22] Matthew Schnaidt. Design, implementation, and testing of MSHN's resource monitoring library. Master's thesis, Naval Postgraduate School, December 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218
2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101
3. Deniz Kuvvetleri Komutanligi 2
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY
4. Deniz Harp Okulu Komutanligi 1
   Kutuphane
   Tuzla, Istanbul, TURKEY 81704
5. Deniz Kuvvetleri Komutanligi 2
   Yazilim Gelistirme Merkezi
   Golcuk, TURKEY
6. Bogazici Universitesi 1
   Kutuphane
   Bebek, Istanbul, TURKEY 80815
7. Chairman, Code CS 1
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101
8. Debra Hensgen, Code CS/Hd 10
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100
9. Ted Lewis, Code CS/Lt 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100
10. Alpay Duman 2
    Zergerdan sok. 7A/6
    Emirgan, Istanbul
    TURKEY 80850

11. Mubeccel Duman 1
Zergerdan sok. 7A/6
Emirgan, Istanbul
TURKEY 80850